VL/01/105

PA 367925

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE

United States Patent and Trademark Office

4

IL01/105

February 28, 2001

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A FILING DATE UNDER 35 USC 111.

APPLICATION NUMBER: ***60/179,926***
FILING DATE: ***February 03, 2000***

PRIORITY DOCUMENT

SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

By Authority of the
COMMISSIONER OF PATENTS AND TRADEMARKS

H. L. Jackson

H. L. JACKSON
Certifying Officer

02-04-00 A/PROV

Box Provisional Application
Assistant Commissioner for Patents
Washington DC 20231

PTO/SB/16 (6/95) (Modified)
EXPRESS MAIL LABEL NO. EL510314798US

PROVISIONAL APPLICATION COVER SHEET

This is a request for filing a PROVISIONAL APPLICATION under 37 C.F.R. 1.53 (b) (2).

02/03/00 C617 U.S. PTO

Docket Number	1283.001PRV	Type a plus sign (+) inside this box >	+

INVENTOR(s)/APPLICANT(s)

Name (last, first, middle initial)	RESIDENCE (CITY, AND EITHER STATE OR FOREIGN COUNTRY)
Rajwan, Doron	ISRAEL
Eyal, Edan	ISRAEL
Yosef, Yuval	ISRAEL
Neerman, Haim	ISRAEL

02/03/00 PTO U.S. $11 60/179926

TITLE OF THE INVENTION (280 characters max)

RELIABLE UNICAST FEC-BASED PROTOCOL DESIGN

CORRESPONDENCE ADDRESS

Schwegman, Lundberg, Woessner & Kluth
P. O Box 2938
Minneapolis, Minnesota 55402

Attn Daniel J. Kluth

STATE	Minnesota	ZIP CODE	55402	COUNTRY	United States of America

ENCLOSED APPLICATION PARTS (check all that apply)

XXX	Specification	Number of Pages	124		Small Entity Statement
	Drawing(s)	Number of Sheets			Other (specify)

METHOD OF PAYMENT (check one)

XXX	A check or money order is enclosed to cover the Provisional filing fees	PROVISIONAL FILING FEE AMOUNT	$150 00
XXX	The Commissioner is hereby authorized to charge any additional required fees or credit overpayment to Deposit Account Number 19-0743		

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

XXX No.

___ Yes, the name of the U.S. Government agency and the Government contract number are: _____

Respectfully submitted,

SIGNATURE Daniel J. Kluth Date February 3, 2000

TYPED OR PRINTED NAME Daniel J. Kluth REGISTRATION NO. 32,146

___ Additional inventors are being named on separately numbered sheets attached hereto

PROVISIONAL APPLICATION FILING ONLY

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re PROVISIONAL Patent Application of: Doron Rajwan et al.

Title: RELIABLE UNICAST FEC-BASED PROTOCOL DESIGN

Docket No.: 1283.001PRV

BOX PROVISIONAL APPLICATION
Assistant Commissioner for Patents
Washington, D.C. 20231

We are transmitting herewith the following attached items (as indicated with an "X"):

X A PROVISIONAL Patent Application comprising:
  X Specification (124 pgs; no claims; no drawings)
  X A check in the amount of $150.00 to cover the Provisional Filing Fee.
X Provisional Application Cover Sheet (1 page).
X A return postcard.

Please charge any additional required fees or credit overpayment to Deposit Account No. 19-0743.

SCHWEGMAN, LUNDBERG, WOESSNER & KLUTH, P.A.
P.O. Box 2938, Minneapolis, MN 55402 (612-373-6900)

By: Daniel J. Kluth
Daniel J. Kluth
Reg. No. 32,146

CERTIFICATE UNDER 37 CFR 1.10:
"Express Mail" mailing label number: EL510314798US
Date of Deposit: February 3, 2000
I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Assistant Commissioner for Patents, Attn: BOX PROVISIONAL APPLICATION, Washington, D.C. 20231

By: _____
Name: Shawn L. Hise

(NEW FILING)

EXPRESS MAIL NO.
EL 523147790US

BandWiz

Provisional Patent Description – no. 1

A System for Efficient Content Delivery in the Internet

1. The Field and Background of the Invention

The patent considers the following environment in the Internet. A Web host of a content provider wishes to deliver content to many prospective users. In the current situation, the content, mostly HTML files, is delivered via TCP/IP, using the HTTP protocol to each user. This leads to several problems. The Web host servers cannot handle the multiple requests, some of them are for the same content. Similarly, the output bandwidth may not be enough to support all requests. Bandwiz suggest a product and technology to handle this problem.

2. A Brief Summary of the Invention

The Product:

BandWiz's modular and scalable product consists of a server-client architecture, as illustrated in Figures 1 and 2. The server (used as a proxy) resides at the content provider site, handles the requests for content and is responsible for organizing and encoding the data for efficient delivery. The client (agent) at the user side, is responsible for decoding and efficient receipt of the data.

Figure 1: The Internet Today

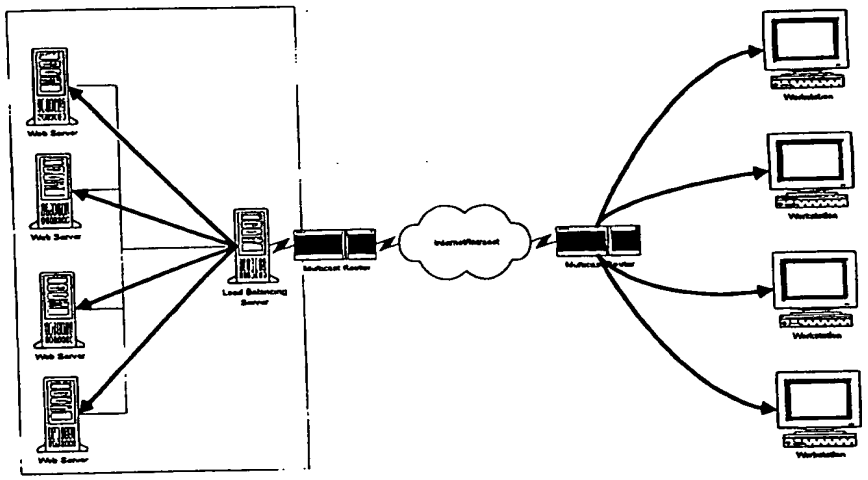
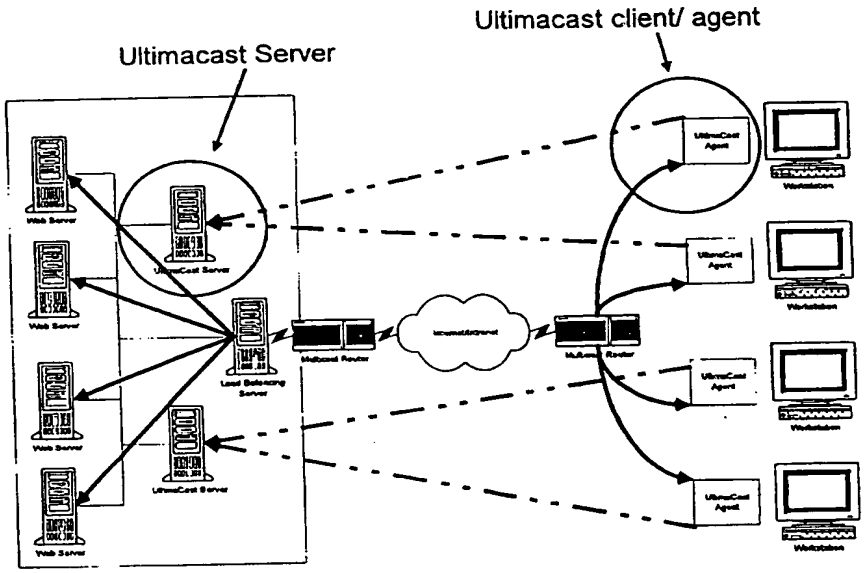


Figure 2: BandWiz Solution



BandWiz solution is most effective in a world where multicast exists. It can accelerate the delivery of large content not limited by the amount of content objects, can be accessed by the entire network and can serve any content provider that gets Bandwiz server. Thus the solution can scale. With this solution the customer will save output bandwidth and equipment, and will be assured of service level agreement.

BandWiz product does not add elements to the net. However, it requires a client at the end user. To shorten time-to-market, an initial release that does not provide all the benefits, is planned . This preliminary product will provide efficient unicast transmission, based on using Bandwiz technology over UDP. Starting with a reliable UDP solution, can ignite the client distribution process so that by the time multicast is available, BandWiz solution will be already in operation.

The Technology:

Multicast can increase data throughput and latency simply by the fact that less date need to be sent as common requested data is not replicated for different users, and by the fact that data may be already transferred at the time a request is made by the user. However, only through BandWiz unique technical solution, multicast advantages can be utilized for accelerating interactive requests that overlap only partially in time and in requested content.

This goal is achieved as follows. BandWiz server organizes the host content in content groups that essentially allow pre-fetching, and encode the data using a special forward error correcting code (FEC). By monitoring content requests it decides to multicast specific data. The data format allows BandWiz's client on the user side to construct its desired content by receiving only a relevant fraction of the transmitted data. The transmission is robust, and the user is not sensitive to packet losses.

Most of the advantages (high throughput, reduced latency) of this end-to-end system are achieved when multicast is available over the net. Nevertheless, some advantages can be achieved in unicast mode. Thus, an initial version of the system can be based on a reliable UDP protocol, and can be deployed before multicast is available. Also, the system can be suggested to accelerate Intranets and other enterprise networks, where multicast can be deployed for the particular enterprise needs.

References:

The product and the technology is described in full details in the document "UltimaCast 4 WWW - System Design"

Reliable UDP Protocol

Before getting into low-level protocol design and description, I'd like us to discuss (and hopefully agree upon...) the basic concept, the requirements from this layer and their implications.

Basic Concept

First of all, let's discuss the use-case. I suggest that we currently bound ourselves to the following simple scenario:

- A Client (=Receiver) requests in HTTP a resource from the Server (=Sender), which is accelerated by the reliable-UDP service.
- The Server initiates a UDP reliable stream to the Client, terminated on successful delivery or fatal error.

Technically, this means:

- "Half Duplex"
  There's no need for "Full-Duplex", so let's keep it simple. For instance, features such as TCP's "simultaneous opening" are therefore irrelevant.
- "Download only, no Upload"
  The initiator is the client, which requests the resource using some other means (e.g. HTTP). The Server responds. Server is always the Sender. Client is always the Receiver.
  Please note: The statistics reported from clients to the server won't be FEC data, and don't need to be reliable, so they would probably need a different mechanism.

More requirements and their consequences

- TCP friendliness
  Basically means a dynamic send rate, which reacts in real-time to the estimated packet loss.

  This probably dictates:
  - Positive Optimized Receiver ACKs
    In order to continuously control the rate according to the RTT estimations. Of course, further optimization should be carried out to fine-tune the Ack rate.
    For instance, should we NACK immediately when we detect loss or should we report couple of losses together?.
    Another optimization here is to ACK when the receiver detects that the rate can be substantially increased.
  - Slow Start
    For inter-protocol fairness. Suggestions like History Retrieval of Previous Connections statistics are unrealistic (resource consuming and low cache hits).
  - Flow Control
    Maintain some sort of allowed unAcked window (some maximum distance between last Acked and last sent packet, like TCP). Window is enlarged with rate. By this we avoid pumping when we get nothing back.

- Streaming support
  The ability to send buffers of a dynamically-built content without waiting for completion.

- API: Non-Blocking as well as Blocking
  Allow Non-Blocking API in order not to force thread per sender (Implementation: Use low-level thread pool mechanism and notified objects ("future", overlapped-I/O-like objects)).

- Bound resources at both sides for scalability.
  E.g., maintain a file cache at server side.

Design Implications

- Use FEC instead of retransmission lost packets.

Send more at the beginning (according to the learnt packet loss at that channel, if any) in order to avoid retransmission.

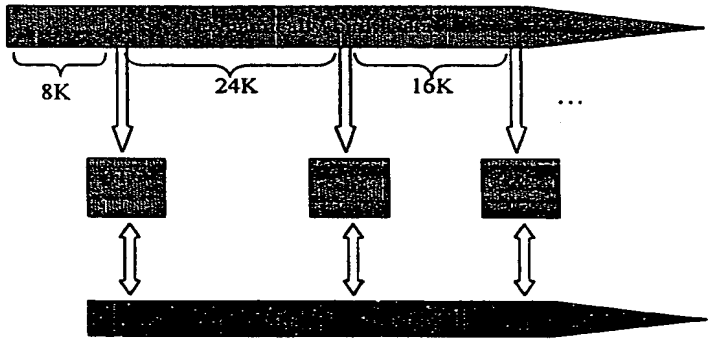
- Advantages:
  - Increase first-time success, and therefore faster delivery when channel suffer from packet loss.
- Disadvantages: standard FEC disadvantages:
  - Sending more before knowing it is really necessary.
  - CPU of encoder (server) + decoder (client)
  - Extra resources (storage) needed for encoded and decoded content in addition to source data at both server and client.

This influences many factors:

- Flow Control mechanism:
  Window should take into account the FEC policy advantages (send redundant packets to increase first-time success probability).

- Basic policy is "Automatic Keep-On Sending FEC" instead of TCP's "Retransmit On Loss Detection". **Loss detection is MAINLY used for rate calculation** (TCP-friendliness), and the retransmission trigger is a side effect. Stream terminates on successful send or on fatal error (e.g. ACK timeout).

- FEC Steaming Engine/Layer (or "On-The-Fly FEC Engine"):
  At both ends (Server and Client) we should have an entity that generates FEC packets for a given data buffer (of a variable size). This enables FEC streaming.



BandWiz Java Client

This document intent is to clear some of the confusion about Java, and the Java client that we might implement in our product.

Java language description

Java language, like C++, is a **compiled language**. The compilation process compiles it from textual source code into a well-defined assembly language, known as Java-byte-code. This assembly language is far superior to X86, Alpha, or any other assembly language exists today. Currently, only emulation of these machines exists, which is called Java Virtual Machine (JVM). In the near future, we will have real Java Machines.

There are several technologies to implement a JVM. The historical one was interpreter technology, which is not in use anymore. Today, there are several methods, like Just In Time compilation (JIT), static compilation (Like C++), and Java HotSpot™ JIT. These technologies execute Java, with all the security, protection, bound checking, and more – in about the same speed of C++. Java is not bounded by C++ performance – a future, well optimized, JIT can run it more than twice the speed of C++.

All the documents that state that Java language is about 70 times slower than C++ are wrong. As you know, I told Yuval that I am ready to bet about it, and gave him a well defined written paper about this bet. If you want, I can send a copy to you.

Modes of execution – Applications vs. Applets

In general, there are two modes of execution for Java programs: the first, is Java application, which is a stand-alone application, like the ones you write in C++. The second is Java applet, which runs in a window inside a Browser, using the **<APPLET>** HTML tag. Java applications, in general, are not restricted. They can use network I/O, disk I/O, and so. Java applets, on the other hand, runs in something called a **send-box**.

Because applets are usually downloaded from the Internet, for security reasons, they are allowed to work inside their send-box, but not outside. They are not allow to perform disk I/O, network I/O, monitor key-clicks, execute native code, and so. There are some exceptions to this rule. For example, they are allowed to open TCP of HTTP connections to the host that they came from, in order to send or receive information.

Digitally signed applets can gain all the authorizations they need. Putting them inside a jar file (or a cab file in Microsoft Internet Explorer) does this. In this case, the Browser will ask the user if he / she allows the Jar file to be installed. If the user allows it, the Java code inside the jar will have the asked authorizations. Another benefit is that this jar will be stored in the client computer forever.

Using Java as a Proxy

In the current design of the UltimaCast-1 system, the client is a proxy agent. In the lightweight version, when running in the same computer, the browser is told that the proxy agent is in address **127.0.0.1**. For this strategy to work, the proxy must be running **before** the browser was loaded. This means that the client cannot be a Java applet.

It can be a Java application, but it doesn't help us. Java applications, for start, need some JVM (which runs outside of the browser). Microsoft tries very hard not to support Java, so they didn't include this kind of JVM inside Windows (although they did include the Microsoft Scripting Host for Visual Basic scripts). This means that we will have to install Sun's Java Runtime Environment (JRE), which takes about 5MB to download.

Second, there was no standard way to install and execute Java applications prior to Java 2 (JDK 1.2). Today, only a small part of the operating systems supports Java 2. We can assume, however, that until we will have a functioning version, most OS will support it.

Using Java for proxy gives us no benefits. We should not use it now.

Using Java applets as "download client"

We can use pure Java in order to implement a "**download client**". For example, when a user wants to download an MP3 song, we can send a page with **<APPLET>**, which downloads the song using Unicast or Multicast.

This applet is NOT the UltimaCast-1 client. It is a small part of it, which includes a modified version of the erasure code. Actually, this is what **Digital Fountain** does, so we might have some non-technical problems here.

This applet has to be digitally signed and trusted by the user for 3 reasons:

1. It should write the file to the disk at the end of the download.
2. It should perform a socket **listen** command, in order to receive UDP packets.
3. We want the browser to store it forever.

This is actually some kind of **advertisement**. We tell the user that we can download this MP3 song faster, and if he / she will install the UltimaCast-1 system, the whole Internet will work faster.

Native Library support for Java applets

We should design UltimaCast-1 system as a DLL, and a thin layer above it. Then, we can allow 3rd party to use it as some kind of connection library, using a business model that we will define in the future.

We can use Java Native Interface (JNI) in order to allow Java applets to access it. These applets should able to connect, and then send and receive information to its host, using multicast or unicast. We should take care not to break the Java security model when doing this.

BandWiz

UltimaCast 4 WWW

System Design

Table of Contents

List of Figures

List of Tables

1. Abstract

This paper contains an in-depth description of **UltimaCast 4 WWW** system. This is "the" technical paper that we have. All the things that the system can and can't do are specified in this document. The implementation of the system will be based on this document.

This paper, among other things, contains a block-level description and interfaces between them. In the near future, even before this document is completed (surly before we agree on it as a whole), we will start to break it into pieces for inner-block design and testing.

I though about using Word's Version support, but it do not work. When having only 4 version, the file grew from 150K to 32M, a factor of 250! Instead, each version will have a name that contains the date I crated it. You can edit the file and send it back to me. The integration is simple: Word has a "Compare Documents" option, in which it will light all the changes, and allow me to accept them or reject them. This will work well unless I totally changed the document until you send me your comments. So, please send your comments quickly, and on the last version of this paper. Please, do not updated embedded Visio drawings, layouts, or move stuff from one place to another. If you have a better way – please tell me soon ☺.

The red, bold ":TBD:" means things that "To be discussed", implying that I do not have a good solution for them right now, or that I didn't have time to finish them.

2. Goals and Requirements

2.1. Commercial goals

2.1.1. Being non-intrusive

The system should be near-non-intrusive, or maybe, non-intrusive at all (to the content provider's side).

2.1.2. Hit counting

We should give our accelerated site an accurate hit counting and bandwidth. This might be that thing that we are paid by.

2.1.3. Time point of presence

We need to give our users benefits even before multicast Internet. Methods like efficient downloading, caching, pre-fetching, differential downloading, and more will do the job.

2.1.4. Bridging the Internet

The Internet will not become multicast at once; maybe it will never be one big multicast area. Instead, we will see bigger and bigger "islands" which are multicast enabled. Each of these islands will allocate multicast IP addresses of its own, using some protocol inside it (Maybe all islands will agree on some version of IGMP, but its not required). Normal IP packets will be the bridges between islands.

We hope that the **UltimaCast 4 WWW** system will be that bridge. I will start with an example: Client **C**, which is inside island **J** wants to connect to host **H**, which is out of **J**. The host **H** should tell **C** about another host, **P**, which is inside **J**. Then, **C** should get information from **P** using multicast. When **H** is updated, it should also notify **P** about this update, since it needs to reflect it. This method can be used as an implementation of a distributed network load balancing ☺.

2.1.5. Teasers

We want to be able to deploy initial version of client & servers in the Internet even before having multicast capabilities in the Internet. We need to give good reasons for users to do that. For example, we can give a Java Applet, which downloads big MP3 files using the Reliable-UDP protocol, and store them on the disk. The user will get 30% less download time, and this will persuade him/her to download the whole client.

2.2. Performance goals

2.2.1. Communication Efficiency

It is common for us to say, "TCP is inefficient by 30% relative to UDP". We need to prove it! We need to give the user a smoother and cleaner browsing experience. We need to use pre-fetching and differential downloading. We need to have higher throughput, and lower latency, while maintaining TCP-friendliness.

2.2.2. Scalability

When using multicast, we want the client to retrieve all the available content without sending even a single packet. In order to do this, the server will need to 'polite' push some amount of information to the client. In order not to create new problems, we should send as little as possible, and work differentially.

2.2.3. Host computer behavior

We need to have low CPU requirements, and memory requirements. We should not create degradation of content that is not accelerated.

2.2.4. Correct allocation of resources

We shall have limited resources in CPU, bandwidth, and multicast addresses. We should use them effectively. In order to do this, we need heuristic decisions. This may be the most complex part of the system. I believe that even being sub-optimal will give us good results.

2.2.5. Suppress cache duplicates

We do not want duplicate content in the cache. Today, the same resource can have different URLs, which causes it to be twice in the cache. You can see it by entering into http://www.cnn.com and http://cnn.com. You will find all the resources twice in

your cache, which implies that they were downloaded twice from the Internet today, there is no way to prevent it.

2.3. Security requirements

2.3.1. Privacy restrictions

We do not want to violate the users right for privacy. We need statistics on the user habits in order to give other users a better browsing experience. We should implement the second without sending even one bit of information that we do not need, in order not to break the first.

For example, the clients will send statistics update. This is OK, because even today when they access HTTP the server get a hit. But, unlike HTTP servers, I want to use unreliable UDP for this statistics. This means that this packet might never get to the server. I do not want to lose this information, so I will make these packets commulative, meaning that each packet contains the information on the previous ones. This means that we will need some kind of client ID. Because we do not want to track our users, we can use a random 160-bit id, per client, per server, per session. So, no one can tell that the same client who seen page **X** at site **Y** yesterday is the client which connects to page **Z** at site **W**. Of course, they can use the IP address, but we do not supply more tracking information!

The user should be prompted at installation time to allow sending of statistics. Declining to do so will result in slightly degraded performance for him (and we should tell him that). Maybe we should give him lower priority when downloading using unicast.

2.3.2. Using cryptographic tools for verification

When thinking about channel coding, the common intuition is that it's adds to the robustness of the system. Examples are parity codes, Hamming codes, Reed-Solomon codes, BCH codes, Trellis codes, Turbo codes, and so.

Our erasure code is different – it enhances errors, rather than reducing them. For example, if we use a simple checksum code, and fail to detect an error in a single

packet, the whole system will collapse. This packet will trash the current resource, and all the files in the Content-Group. Then, it will trash every Content-Group that has a common file with this one. If, for example, each page has a menu on the side, and one image of this menu has a bad block – it will trash all text and data of **all the pages in this host**. This effect will never go away. The only way to fix it will be to uninstall and reinstall the UltimaCast 4 WWW system ☹.

In order to prevent it, we need a strong, secure code. This is why we should use SHA digests. If the URLs in the HTTP server will be left unchanged, we might need some kind of hash-table to map them into SHA digests. Failing to keep it up-to-date will cause displaying of older version of the text, as happens today. But it will not have the magnifying effect that I described above. Also, it will never download the same page twice.

2.3.3. Using encryption for paid pages

We need to be able to support accelerated paid pages. This means that we need to be able to download the pages, and send them only to registered users. We want to give the same security as HTTP. No more, but no less.

Yuval: Do we need to support the pay-per-page business model, or just the register-users-have-it-all model?

:TBD: Think about it, and add it somehow to the description below.

2.4. Engineering considerations

2.4.1. Interoperability

We want to design a file format that can be used as-is in different computer architectures. We want robustness, version control, and some kind of backward and upward compatibility. This file format should allow memory-mapped files too.

2.4.2. Installation

We should have a clean install and uninstall.

3. System Overview

3.1. System Block Diagram

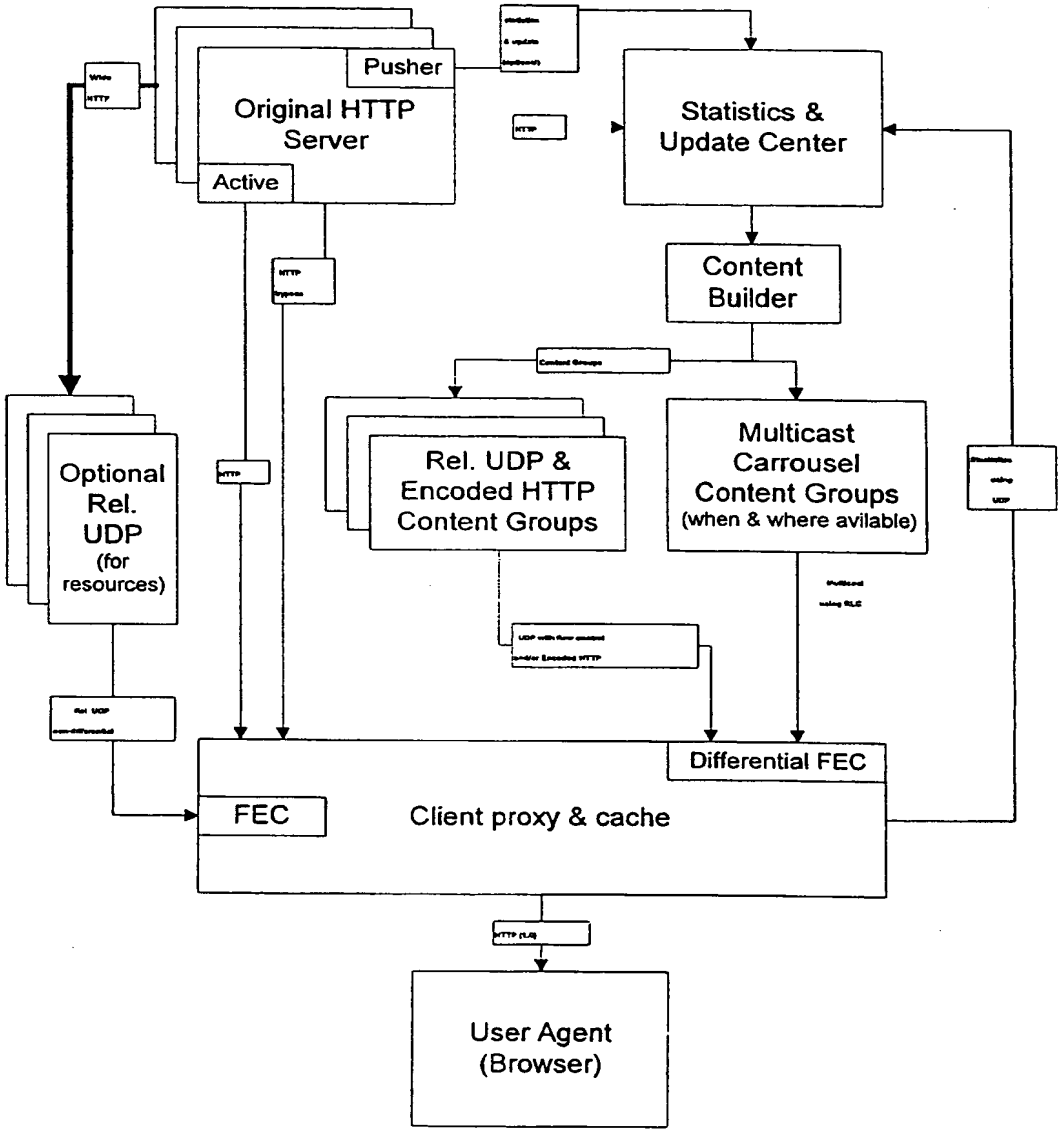


Figure 1 - System Block Diagram

3.2. Quick explanation

Each block is a system component, which can be in a different computer. The "3D" blocks are blocks that we might need a cluster ("farm") of servers in order to implement for high-capacity host.

Each line is a connection between blocks. The connection protocol is written on the line itself.

Only the "Client proxy & cache" block is the "client". Other components are the "server". The client gets information from the server, and feed it back with statistics.

4. Common scenarios

4.1. Location of each component

:TBD: The minimal configuration of the server and client. The maximal configuration of the server and the client. Integration with the ISP.

4.2. Original HTTP site

:TBD: Modifies a lot. Dynamic HTTP. Static HTTP. And more.

4.3. User behavior

:TBD: Some common user scenarios. First time, after a month, after one hour, organization cache, cache in ISP, and more.

4.3.1. First connection ever

The user types a URL which links into a specific host for the first time ever. Our proxy gets this request, and returns it by HTTP. Then, it starts a setup process. The user should not get degradation in the download of the first page.

First, it needs the **ACTIVE** URL content. This is the boot information for the whole process. The client connects to **http://host/__UltimaCast4WWW/ACTIVE**. If this URL does not exist, this host is not UltimaCast 4 WWW accelerated. Our proxy will not check this site again, for at least 24 hours, or until a restart.

If this URL exists, it will contain the **ACTIVE** record, optionally followed by the Master-Content-Group-Header itself. This is a little tricky protocol. We should use features of the HTTP transport to get what we need, and to save RTTs.

First, the server may respond by sending an HTTP redirection. We need it in order that the original site will only have to add a fixed URL that never changes. Then, our server will handle the actual request.

Another thing is that we want to save RTTs when obtaining the Master-Content-Group-Header itself. This is why we join it to the end of this connection. The problem is that it will send unneeded data in the case that the client already has the up-to-date information. In order to solve this, we shall use the HTTP "if-modified-since" header. The server will have it's own logic to decide if it wants to return it or not.

First, the client should download the **ACTIVE** and remember it. If the server will also send the Master-Content-Group-Header itself, the client should download it and store it in its cache. If it is already exists, it should terminate the HTTP connection immediately; there is no way to get out-of-date data because the client do get the SHA of it.

Second, the client needs to check if it already has the Master-Content-Group-Header in its cache. If it was appended to the header, the client surly has it. If not, the **ACTIVE** contains information about its SHA and another IP addresses, which enables the client to download it using the combined protocol. It can start with multicast, fall back to unicast, and then to HTTP. The client should store it in the cache, for the next time. The amount of data is expected to be very small – about 50 bytes per Content-Group. This gives us about 5KB for a big host. The client should also remember it in-memory.

At this stage the client still does not have the required information to work. The client needs the Content-Group-Headers themselves. So, third, the client enters the download of Content-Group process for this Master-Content-Group. Each downloaded Content-Group-Header will be put in the cache, as always.

When this 3-stage process ends, the client is ready to do its job as a proxy. It has all the Master-Content-Group in-memory. The size of it is about 100KB to 300KB for a big and pushy host. A non-pushy host can send less data, even if it's very big.

Then, when the client gets requests to this host, it can compute the SHA of the requested URL and check if it has non-expired URL-Connection for it. If it does, it can use it. If it was expired, it should access the **ACTIVE** URL again, and get a new expiration time. This is all a part of the combined protocol.

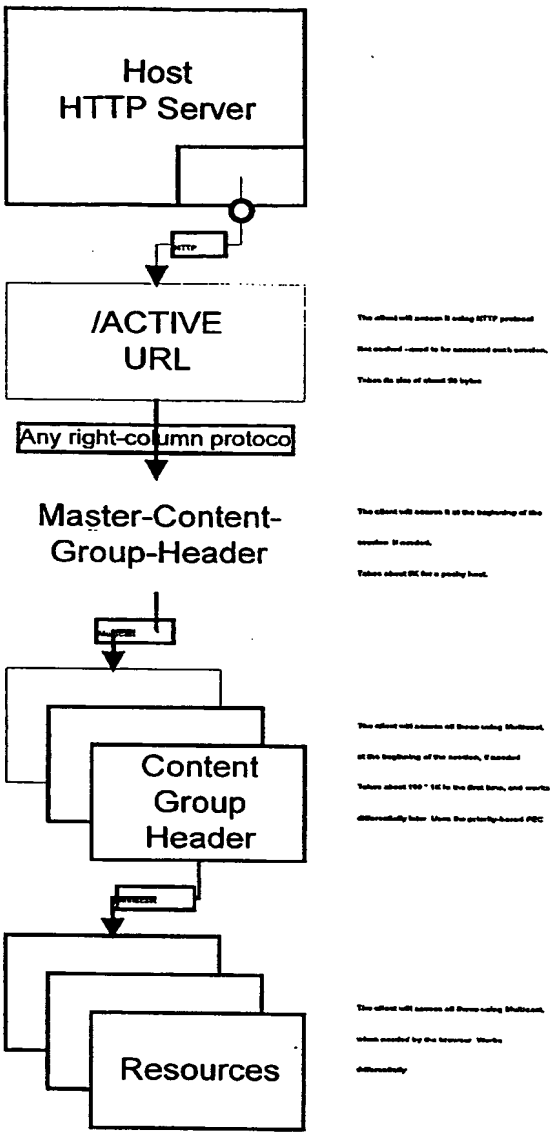


Figure 2 – Push Information Hierarchy

4.3.2. Connection after one day

In this scenario, the clients connect to the main page of the site that it visited yesterday. Some of the hot pages in the site were updated, but most of them were not.

First, the client will check the **ACTIVE** URL. This URL will contain different Master-Content-Group-Header, so the client will download it in the same HTTP connection, or in multicast. This will take the full 5K bytes. Then, the client will download the Content-Group-Headers themselves. Because most of them were not updated, the client will receive only 5K instead of 100K. At this point the client is synchronized.

The user agent asks the client to get the latest news page. The client will download this page using multicast, but it will add equations for most of the resources in this file. This process is very efficient, and will happen without sending even one packet.

5. Module detailed description

5.1. Original HTTP Server

We should be as non-intrusive as possible. All we want are three things:

1. A URL which will be the "root" for everything.
2. Statistics for the normal HTTP hits in the site.
3. Notifications about important pages that were updated.

We can do well without the last two. The first, however, is a little bit tricky. It is better to have it.

:TBD: **Edan, Omer**: We should think about Cookies. They will cause a big problem!

5.2. Statistics & Update Center

Will get information from these sources:

1. The statistics that the clients sends.
2. The statistics that the original server sends (optional).
3. Detect updates in the original site, by polling.
4. Notifications about updates in the original site (optional).
5. Administrator command.

Based on all this information, the statistics center will decide about building Content-Groups, and transmitting them over Multicast and/or Reliable-UDP and/or HTTP. Then, it will issue these commands to the Content Builder.

:TBD: Describe it a little more.

5.3. Content Builder

Builds a content for a given URL. This will recursively detect all the embedded resources inside a URL, and pack them in a Content-Group. This builder needs an

HTML parser, compression utilities, and more. :TBD: **Yuval**: Should we buy these tools now?

Then, depending on the command from the statistics center, it will deploy these Content-Groups to the multicast carrousel and/or to the Reliable-UDP server cluster.

:TBD: Describe it a little more.

5.4. Multicast Carrousel Content Groups

This is a stupid data carrousel. It does not accept packets from anyone, except from the content builder. I think that a single PC will be able to serve all the hot pages in the entire site. We need the multicast to be deployed for this component.

This data carrousel should be able to get hot updates for multicast groups without effecting other groups at all.

:TBD: Describe it a little more.

5.5. Rel. UDP & Encoded HTTP Content-Groups

This is a server cluster for serving clients using reliable-UDP and Encoded HTTP Content-Groups.

:TBD: Describe it a little more.

5.6. Reliable UDP for resources

If we want to use the Reliable UDP for serving large resources, we need this component. It does not need to get statistics or anything like it. It should simply get requests for URLs and serve them. This component is very heavy. It will need a lot of CPU and a cache (disk or memory) in order not to download too much from the original HTTP server.

:TBD: Describe it a little more.

5.7. Client proxy & cache

This is the client proxy. It can be in the user's computer (lightweight), in an organization outside/with the firewall, or even in the ISP office. It is used as a proxy. It should have a large disk space for caching. It should work with all protocols. It should have FEC. It should load when the computer boots. It should have a GUI for configurations.

I think that this proxy should be HTTP 1.0 to the client, but Edan thinks differently. When working with HTTP 1.0 the browser will open several concurrent connections, rather than using the same connection for several resources. I think that the first method is more efficient, because it allows us to return any of the resources and return others. The advantage about not creating TCP connections disappears because the proxy and the browser are near (on the same computer or LAN). Please notice that Internet Explorer 4.0 and 5.0, by default, do not use HTTP 1.1 through proxies. I think that it's like this because they checked it, and found out that this is more efficient. Another reason might be that proxy for HTTP 1.0 fails if connecting to them using HTTP 1.1. This does not happen to HTTP to server, but maybe the proxy protocol is ill defined.

This client will be free. I will buy everyone a drink when we will have 1000 users who downloaded the client ☺. Anyway, this is a beta of the specification, and may change without notice. In this case, I will not have to buy everyone a drink ☹.

:TBD: Describe it a little more.

5.8. User Agent (Browser)

This is the user browser, or whatever. We should configure it to use our client as its proxy.

6. Modules Connections

:TBD: Describe each of the lines in the drawing. I guess that the protocols for the client are more or less defined. I need to define, for example, the protocol of the statistics update from the client; or the protocol from and to the content builder.

7. Protocols

There are 6 new protocols. Each of them needs different resources, and has its advantages and disadvantages.

	With Content-Group-Header	Without Content-Group-Header
Multicast	FEC: Yes Code: 50:50, random, iid Differential: Yes (common files) Pre-fetching: Yes (content-group) Flow: RLC Init: By Statistics Center Client: Does not send data	FEC: Yes Code: Mostly systematic Differential: No Pre-fetching: No Flow: RLC Init: By Statistics Center Client: Does not send data (Need to send some side-information packets all the time)
Reliable UDP (Unicast)	FEC: Yes Code: 50:50, random, iid Differential: Yes (common files) Pre-fetching: Yes (content-group) Flow: Client sets speed Init: By Client, UDP Client: start, stop, and set-speed.	FEC: Yes Code: Mostly systematic Differential: No Pre-fetching: No Flow: Client sets speed Init: By Client, UDP Client: start, stop, and set-speed. (Need to send some side-information packets at start)
HTTP as Transport	FEC: Yes Code: 50:50, random, iid Differential: Yes (common files) Pre-fetching: Yes (content-group) Flow: TCP, retransmissions Init: By Client, HTTP Client: HTTP & TCP protocols.	Uses the old HTTP (no-FEC, no-code, not differential), or, sends Content-Group-Header, and then the data, like the block on the left side.

Table 1 – Basic Protocols

7.1. Basic protocols – Left column

7.1.1. Multicast, with Content-Group-Header

This is "the" protocol. The whole system is build in order to get to this block. At this block, the clients do not send even one IP packet to the server. This means that the server can handle any number of clients, using a very simple data carrousel.

In order to connect the client needs a Content-Group-Header, and the IP address of the multicast group which serves (only) this Content-Group.

TBD: **Edan**: We need to define the application-level interface to this protocol.

7.1.2. Reliable UDP, with Content-Group-Header

When no multicast addressed available, or in a non-multicast region of the Internet, the most efficient protocol is this one. It uses the Reliable-UDP protocol that we shall define, with the ability to add equations for files that the client already has.

The server can be relatively simple. It can get Encoded-Content-Group, and send its packets to a given IP at a given speed. It uses the same ACK / CHANGE-RATE / DONE commands defined in the Reliable-UDP protocol for resources below. There are several differences, however:

1. This server gets encoded packets. Rather than encoding at runtime, it serves them in a data carrousel, which should get into a second round in rare cases.
2. The code here is 50:50 random. The code for the server below is mostly systematic.
3. This server supports relatively small files. The server below supports huge files.

In order to connect, the client needs a Content-Group-Header, an IP of the Reliable-UDP server that serves this Content-Group (among others) and a well-known port number of it. The client application-level interface for this protocol is complex. The client should give the protocol a cache object, and name of the Content-Group:

accept-resource(<like below without URL>, C.G.NAME, cache)

The Content-Group name is well defined in this document, and must exist in the cache. The protocol must first run the verification process on it, to detect valid resources and missing ones. If all valid, the communication will not start. Otherwise, it should build a FEC data that contains all the added equations, and start the communication process.

In the communication process, each packet received is checked using an application-level callback. It might be a packet that causes one or more resources to be decoded. In this case, it will send these resources to the cache, which will invoke listeners on it (if any exist). It will continue to receive packets until all resources are completed.

7.1.3. HTTP as Transport layer, with Content-Group-Header

Sometimes clients cannot use the UDP protocol. For example, a client behind a firewall, a PDA, or a **HAVi** device might not have UDP access. In these cases, we can still have the pre-fetching and the differential downloads using HTTP. We simply put the data packets to HTTP connection, and the client reads until it has enough. I hope that we will not need it, because Reliable-UDP is more efficient, and I guess that it will take fewer resources in the server side.

In order to connect the client needs a Content-Group-Header, and the IP address of the HTTP server that serves this file (among others).

7.2. Basic protocols – Right column

7.2.1. Multicast, without Content-Group-Header

This is what **Digital Fountain** does. This protocol is needed for downloading a single, big file, from the server to a lot of clients simultaneously. It does not allow us to work differentially (by adding equations), or pre-fetching. However, it allows us to send huge files, with systematic equations.

It is good just for one thing: send big files to a lot of users. It is feasible to use Java Client in order to collect the file and store it on the user's hard drive. This format is not suitable for sending small files, or for sending HTML pages.

:TBD: Nadav, Meir: We cannot use systematic equations and RLC. How does Digital Fountain code works? How can it handle the flow-control?

In order to download files using this protocol the client need to know the IP address of the multicast group. The client also needs to know the size of the file, and its name. This information can get to the client using meta-data packets from time to time, or by the same one who told the client about this IP address in the first place. :TBD: **Edan:** We need to decide about it.

7.2.2. Reliable UDP, without Content-Group-Header

This protocol is needed for downloading a single, big file, from the server to the client. It is relatively simple, but we need to tune it carefully. It does not allow us to work differentially (by adding equations), or pre-fetching. However, it allows us to send huge files, with systematic equations.

The reliable UDP can be used as a transport layer, like TCP. There is a major difference, however: TCP (like all other layers in OSI and other communication models) is stream-oriented. When TCP connection is established, the sender has a pipe, which it can put data into and the receiver will get it. The reliable UDP, on the other hand, works in the resource level, when a resource is an array of bytes who is known to the server when opening the connection. The protocol I suggest here does not support compression, sub-files, or adding equations for parts of the file, **meaning that we cannot use it for downloading the Content-Groups.** This protocol fits only in the block that I called "Optional reliable-UDP for resources", and for downloading resources using a download-client, like **Digital Fountain**. We can still use it effectively, for example, in a Java Client that downloads a large MP3 file.

In order to connect, the client needs the resource URL, an IP of the Reliable-UDP server that serves this URL (among others) and a well-known port number of it. The client initiate the connection by calling a non-blocking method:

accept-file(server-IP, server-port, client-port, initial-rate, URL)

Than, the client sends a single UDP request packet to the server (and some re-transmission mechanism if this packet is lost). When the server gets this packet, it

should check if the URL exists. If not, it will send a single file-not-found notification packet to the client port. If it does exists, it will load it (or map it) to memory, and call the non-blocking method:

send-file(client-IP, client-port, URL-data, initial-rate)

Then, the packets will travel from the server to the client, at the initial rate. The client should send ACK command from time to time, DONE command when it has enough packets, CHANGE-RATE command when there are too many or too few loses, and so. We should use some of the ideas in the RLC code ☺. The server should terminate the connection if the client sends DONE, or does not send any packet for some time.

The server should start by sending a packet that contains this information:
Block-Header – A well-defined 64-bit header.
U32 – The size of the resource, or –1 for file-not-found.
URL-Content – Describes the content of the requested URL.

In order that this information will not get lost, we can, for example, send it 3 times to the client, or, add a re-send command for it (while the client holding the packets until we get it without being able to decode them). The client needs this information for several reasons:

1. Version control. This ensures that the client and the server are using the same version of the FEC and so.
2. Size of the resource – in order to know how many packets it contains. This is critical for the decoding.
3. Verifying that both sides are talking about the same resource.
4. Verifying that when the transfer ended, the client has the correct file.
5. Allowing a smart re-connect, or **resume** option in the client ☺. The client should know that this is the same resource for resuming.
6. Allowing the client to detect that it already has the file, and abort the download before it starts.
7. Allow knowing the expiration time of this resource.
8. Allow knowing the mime-type, and more meta-information, in order to send this file to the user agent (browser).

The above list implies that the client should get this header immediately, and have the opportunity to allocate buffers, or deny the download (if too big, bad version, or if already exists).

Only when the connection is ended the server is able to de-allocate its memory buffer. The client should get a notification at this stage, with the header information and the suggested bit-rate for next time. Only then, the transport layer can free the resources.

From the programming point-of-view, we should create a single code for both the Reliable-UDP protocols. Supplying hooks that are strong enough to do anything we need should do this. The protocol should be very simple and deterministic to implement, except a "black box" which gets all the relevant data about packets accepted and loses, and determine the connection speed.

This "black box" contains all the magic in this protocol. A good implementation of this box will give a robust and working protocol. By isolating it, we can develop all the system around it, and fine-tune this box at a later stage. As far as I can tell, even a "return 33600" implementation will do the job at the beginning.

The server should not remember any information about a specific client, or be able to identify clients in any way. It should only be aware of ongoing sessions. The client should remember information like the connection speed, and give it to the server thought the "black box" as the initial rate.

One of the complex things that the "black box" should do is concurrent connections support. For example, if we are connected through 33k connection, and the client downloads a file at 30k, and new sessions starts, should we send speed change to both the new and old connections notifying that the speed should be 16k? Maybe it is better to wait that we will loose packets, and then change the speed (as TCP does)?

7.2.3. HTTP as Transport, without Content-Group-Header

Same as above, but when we do not have the Content-Group-Header. The solution is simply send it first (which works, because TCP is reliable), and then put the encoded

packets. I hope that we will not [redacted]d it, because Reliable-UDP is more effi[redacted] and I guess that it will take fewer resources in the server side.

In order to connect the client needs the resource name, and the IP address of the HTTP server that serves this file (among others).

7.3. Tricky protocols

7.3.1. Stateless Reliable-UDP

In general, the Reliable-UDP is a statefull, complex protocol (although my guess is that TCP is worse). The server should remember each client's IP and port, which resource, the rate, RTT approximation, and much more.

But, we can use a little trick in order to send small files. We need it, in order to send Content-Group-Headers to the client. For example, the server wants to send a very large Content-Group-Header, which takes 1800 bytes. Let's say that we use 512 bytes per packet. The server will break the file into 4 packets, and encode them into 7 packets. Then, it will put all the packets on the Internet.

There is a good chance that at least 4~5 of these packets will reach the client. If they did – we have saved a huge amount of resources in the server. If they didn't, the client will ask again, and the server will send another 7 packets. Any 4~5 of these 14 packets will allow us to decode this resource.

I think that even in this second case we have positive savings. Starting a protocol, sending packets, waiting for all the packets to get, and then send a termination packet (which might get lost too) is much more complex (and takes more RTT) than this stateless protocol. :TBD: **Meir, Nadav**: Do you think that it's worth a patent?

We can even go a little more, by implementing an outgoing queue in the server that supports delayed packets. In this case, the server can put a lot of packets in this queue, and forget about it. If the client didn't get enough information, it can always ask for more, and combine the equations ☺.

7.3.2. Differential Reliable-UDP for resources

We can add differential download to the Reliable-UDP for resources protocol. I will ignore the problem that in the current description the code is mostly systematic, and assume that the code is random, 50:50, iid. So, for differential downloading all we need is a method for the receiver to add equations using its information in the cache, and this is what I will describe in this section.

One solution, is using the **rsync** protocol as-is. In this way, the server breaks the file into block, when a block is a single packet or a fixed number of packets. The server should send the SHA digest and CRC-32 of each block to the client. These 24 bytes of information allow the client, using a lot of CPU, to find matches with the current version it has.

There are two problems with the **rsync** solution. The first is that the client computer should have a lot of CPU power and lots of memory. The second problem is that the information about the blocks will take a lot of place. For example, a single MP3 file will take: 4M / 512bytes * 24bytes = 192K. This is too much. We can use bigger blocks, but then we might be able to add fewer equations.

The second solution is that the client will tell the server which version of the file it has, by sending the SHA of this file. Then, the server checks if it has this previous version. If it doesn't have it, it ignores it. If it does have it, the server will send a small amount of data, maybe even a single packet, with all the information.

For example, the client will say "please send me URL X, when I have a version of this URL with the SHA of Y". Then, the server can reply "this is the current version". The differential reply can be "I am going to send you the newer version, which has SHA of Z; But before it, please cut from offset **off1** and create **n1** equations from **packet1**; cut from offset **off2** and create **n2** equations from **packet2**; and so".

This is the most efficient way there is when there are only a small number of alternatives. The server will do batch computation of all the differences, using **rsync**, and send the results to the client. This method is preferred.

A third, very powerful solution can be used for multicast. The server will send the resource in multicast. But, from time to time it will send packet containing a list of statements like this one: "if you have file with SHA of X, please cut from offset **off1** and create **n1** equations from **packet1**". The client will check it's cache, and if the file exists, it will add the equations accordingly.

7.3.3. Reliable UDP for streams

A future protocol that we can design is Reliable-UDP for streams, instead of resources. This will allow us to send a stream from side to side, like TCP does. It should have the exact interface of TCP. In general, it will be a new implementation of TCP, with some differences:

1. Unidirectional. There is a sender and a receiver. (We can make it two-way).
2. More efficient throughput, maybe by 30%.
3. Less up-stream packets. Better for asymmetric connections.
4. A far higher latency. I am talking about a magnitude here.
5. Compression?
6. Security?

In Edan's view, this protocol is block-wise version of the normal protocol:

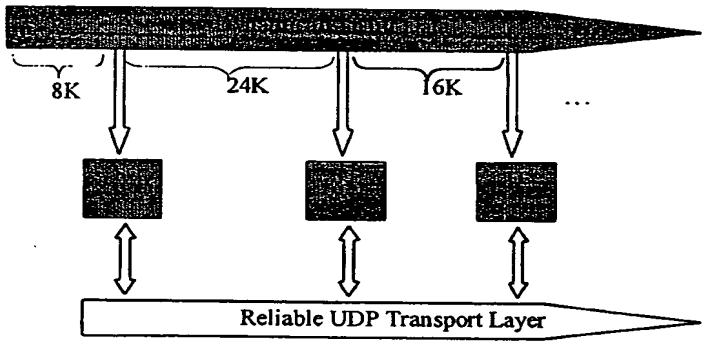


Figure 3 - Reliable-UDP for streams, Edan's view

I think that it's a little more complex, because we do not want to "pay" a whole RTT for each small block, and get a header for each part. Instead, the client should tell the server the speed, the block size, and code-rate. For example, send me at 56kbps, with 100 systematic data packets per block, plus 5 packets of data correction. Than, a very

smart "black box" in the client should decide about each of the packets. The client should send a notification per block. The notification can be that the block is done, and transferred to the application layer. Another notification is "please send me more N parity packets for this block".

If we implement it right, this can be a really state-of-art protocol, which will enable to download and play MP3 files directly, if the bandwidth is high enough. We (the current group) can design and implement this protocol, but we can use some help for the design of the "black box".
:TBD: **Meir, Nadav**: This is surly worth a patent. Should design it and patent it?

There is one good commercial reason not to use it. This will be a feature that we cannot support when going multicast. This means that our users will see degradation when the multicast opens. This might not be a good idea. **Yuval**: What do you think?

7.3.4. Downloading resource – The combined protocol

I thought about some kind of server-based mixture between the protocols, which the statistics center decides on. The idea is simple yet powerful.

The client needs to download a resource that belongs to a specific accelerated host. First, it checks whether it has the Content-Group-Header for it or not. For the most common resources it will have a Content-Group containing it, because the server will push it to the client.

If it does have a Content-Group-Header for it, it also has a Resource-Download-Info, which contains up-to 3 IP addresses for the 3 protocols. The content provider should decide which protocols are valid for each Content-Group. If more than one way is given, the client should try the multicast first, then the unicast, and finally use the HTTP transport layer. In all cases, there is no need to re-download the Content-Group-Header itself; only the data is needed. Please notice that the packets in the 3 protocols are the same. The decoder can get packets from all of them, and use them in the decoding process.

Failure occurs when:

1. The IP address is set to zero.
2. The protocol was disabled in the client.
3. The server does not respond, or stopped responding in the middle.
4. The Content-Group validation process fails after the download.
5. Any other error.

When using HTTP protocol, the client needs IP address, port, and path. If the IP address in this locator is 0, the Content-Group cannot be accessed using HTTP. If the IP is -1, the IP is taken from the original URL that connected us to this host (we need to define the exact semantics for it). Otherwise, this field contains the IP address to use. The port is always 80. The path is some prefix combined with the Content-Group name. For example: **/__UltimaCast4WWW/ecg/012...DEF**, when "ecg" stands for Encoded-Content-Group, without the Content-Group-Header first.

In order to get the Content-Group using unicast (Reliable-UDP), the client needs an IP address, as defined above. Then, it can access the server at a known port, and ask for the Content-Group. In order to get the Content-Group using multicast, the client needs an IP address of the multicast group. Nothing else is needed.

On the other hand, if the client does not have the Content-Group-Header, it still knows to which host it connects, and has the **active** information. Inside it, it can find default address for Reliable-UDP server. Then, it will send a single packet to this server using UDP. This packet will contain the needed URL, as defined above. The server may:

1. Send a single packet, notifying that this file is not accelerated at all. In this case, the client will use the old HTTP for it. This notification may or may not contain the SHA of the resource in that URL. If it does contain SHA, and the client has a file with this SHA, it can use it.
2. Send the requested file using Reliable-UDP, as defined above. The first packet will contain meta-information about the file SHA and size, so the client can abort the download if it already has this file.
3. Send a single packet, notifying that this resource is transmitted using multicast, and give the IP address of the multicast group. In this case we can also send SHA

and [redacted]. This allows us not to send meta-information when sending the packets in multicast.

4. Send a single packet, notifying that this resource should be accessed using encoded HTTP which contains the Content-Group-Header, as defined above.
5. Send a Content-Group-Header that contains the file using Reliable-UDP. Than, using this Content-Group-Header the client will execute this combined protocol from the start. Since this file is small, we can use the "Stateless Reliable-UDP", as defined above.

In any of the cases, the client might fallback to the old HTTP protocol from the original server.

8.1.1. SHA-1 (Secure Hash Algorithm)

SHA-1 is an open-source digest algorithm from NIST. It accepts an array of bytes (actually, bits), up-to 2^64 bits long, and convert it into 160 bit signature. This algorithm is highly efficient for software implementation, since the 160-bits are actually 5 32-bit registers, with only register level operations between them.

This algorithm, unlike CRCs and so, is cryptographic-secure. This means, that one cannot feasibly find a file that their SHA-1 digest is X. One cannot create two files with the same digest. There are (almost) no collisions. We need all these features of the SHA-1 algorithm.

SHA-1 is superior to the obsolete original SHA algorithm, which had a flaw. In all places that I say or write "SHA", I mean "SHA-1".

8.1.2. FEC (Forward Error Correction)

A FEC is an Information-Theory base technique, which allows to send encoded information, and recover it in the client, without specific requests for retransmission. I will give a short description of it, without getting too deep.

We want to transmit a file. First, we break the files into N blocks. Each block is exactly in the size of the IP packets that the transmitter will use (maybe 512 or 1024 bytes). The encoding process uses a seed for pseudo-random bits, and generates N bits, corresponding to the N blocks. Then, it uses bit-wise **xor** on the blocks, which the bit is set. The resulting block is sent over the Internet, with the seed, and a checksum. Then, it does it again, infinitely.

The receiver collects packets. It validates the checksum. Then, it calculates the N bits from the seed. The receiver collects N+e packets, until the matrix with N column and N+e rows is in the rank of N. Then, it can inverse the matrix, and resolve the code.

We can show that if the bits are **iid**, with 50% to be one (50:50), the expectation of **e** is about one. This means that in the normal case all we need is one extra packet in order to decode the data.

The client needs some side information. It needs a version – to validate that the transmitter and the receiver uses the same FEC. It needs the SHA of this resource in order to validate it at the end. It needs the size of the resource in order to know N, and in order to build the file without padding zeros.

The real world is a little more complicated. We need the decoder to do the job sequentially, which means, that it should do some small amount of work each packet, rather than waiting to all the packets and then blocking the computer.

Sometimes the client has some of the information in the encoded file. For example, when this file is a Content-Group and the client have some of the files. Other example can be any form of differential downloading. In these cases, the client should be able to add systematic equations to the decoder. These equations should be handle efficiently.

It also might be able to get some of the blocks before others. We need each block immediately. This will enable the application layer, using Content-Group-Headers, to decode some files before others. We need it for the FEC that has information in different priorities.

:TBD: **Nadav, Edan**: We need to define the exact interfaces of the encoder and decoder to support all these requirements.

8.1.3. FEC with priority-based information

:TBD: **Nadav, Meir**: Please review this section.
Sometimes we want to send information that is more important with information which is less important. We need it when sending the Host-Mater-Content-Group to the client. This Content-Group contains files, ordered by importance, when the most important files are first. This method works even better if these files also get updated more frequently.

Instead of encoding 50:50 iid, we encode not 50:50, not identical, not independent. We break the code into priority blocks. For example, a 100-packet Content-Group will be broken to higher priority **P1=[1,8]** that contains packets 1 to 8, medium priority **P2=[1,24]**, and low priority **P3=[1,100]**. The encoder will encode, for example, **25%** of the packets from **P1**, **25%** from **P2**, and **50%** from **P3**.

In the first time a client connects, the expectation of the number of needed packets is:

1. For **P1** – (8+1)/0.25 = 36 packets.
2. For **P2** – max[(24-8+1)/0.25 , 36] = 68 packets.
3. For **P3** – max[(100-24+1)/0.5 , 68] = 154 packets.

In the normal way, it would take 101 packets for P1, P2, and P3. Some of the files get to the client 3 times faster, while others gets 1.5 times slower. It hard to tell whether this is better or worse. It depends on how much higher the priority of the resources in P1 relative to the last resource. My guess is that it's much better.

Worst-case scenario is that **P1**, **P2** were unchanged, and **P3** was changed. In this case, we will need twice the packets to resolve **P3**. For this method to work efficiently, we need another assumption: The more important the page is, it will get updated more frequently. This assumption gives a low probability on this scenario.

Best-case scenario is that **P1** contains much more changes than **P2**, and **P2** contains much more changes than **P3**. In this case we shall need the same number of packets, and we even might get **P1** sooner!

We need to check if this method works. It is surly worse than allocating 3 multicast groups and subscribing to them independently. But if we have limited resources, this method might help. After receiving only **36k**, instead of **101k**, the client is able to connect to the hottest **8** pages in the site without sending even a single packet ☺. Please note that it can contain sooner, but with sending a single packet to Reliable-UDP host.

Another important thing is that we can ensure that the client needs to get **P1** if it needs **P2** and **P3**. We do it by changing the order of the resources each time a resource is

updated. So, the resource order reflects both their importance and the update time. This is a tradeoff that we need to think about.

TBD: **Omer**, if you want to use UDP ports, this is the place. We can use 3 ports, and can stop listening to port **P1** when we have the information in it.

8.1.4. Field size, byte order

In order to support interoperability, we shall not use C++ **int**, **short**, **long**, and so. These are not well defined in two aspects: one, is the number of bits in them, and another is the byte order (little/big endian). In this document, we will use:

U8 – Defined to be unsigned 8-bit integer (0 to 255).

U16 – Defined to be 16-bit unsigned integer (0 to 65535), in the Network byte order.

U32 – Defined to be 32-bit unsigned integer, in the Network byte order.

All the file formats are in the Network byte order, formerly know as Motorola byte order (MSB first). We shall not use floating-point numbers inside files. All file formats should be defined such that each data size is aligned to its boundary. This means that **U16** should be at even-bytes offsets and **U32** in quad-bytes offsets.

The **ACE** package has the same definition in different names. For the implementation, we shall use ACE. However, we need support for both streaming (a read command which swaps the bytes if needed) and memory-mapped files.

8.1.5. Block-Header (for both blocks and files)

All the files (and blocks inside the files) that we use shall have a common 64-bit header. This will allow us to nest blocks in a robust way. This header should be aligned on 64-bit boundary on all circumstances.

The header is:

U16 – Type of file. We shall use a common definition file for the whole system.

U8 – Major version number.

U8 – Minor version number.

U32 – The size of this block or file, including this header.

Every program must validate the type of file it reads. If it does not match the expected type, the program must halt, and report it.

In general, the major file version is for incompatible changes, and the minor is for compatible ones. When a program that designed for a specific version reads a file, it should check the version. If the major version is bigger than expected – this file cannot be read. If the major version is smaller than expected – the program should decide if it wants to convert it or not (backward compatibility). If the major version is as expected, the program must read the file correctly, and the file should contain information depending on the minor version number.

For example, version 1.0 defines a bit-field, which includes some reserved bits. A writer for version 1.0 must set them all to zero. A reader for version 1.0 must ignore these bits (forward compatibility). Let's suppose that version 1.1 give meaning to some of these bits. A writer for version 1.1 must set these bits to their correct values. When a reader for version 1.1 read version 1.0, it should understand that these bits do not have a value (backward compatibility). In order to achieve this the easiest thing to do is usually to convert all previous versions to the latest version. This is easy to implement with streaming, but a little harder for memory-mapped files.

The block size is needed in order to skip known or unknown blocks, and integrity checks.

8.1.6. Resource

A resource is single file on a disk, byte array, or whatever you call it today. The name of each resource is its 160-bit SHA digest, which depends only on its content. It's name uniquely defines it's content, for all practical purpose.

A resource **does not** have a file-extension, a mime-type, a date, or any other meta-information about it. These will be added, if and when needed, by higher level definitions.

8.1.7. URL-Content

This structure links between a URL and it's content, and defines for how long this link is valid. For any resource that we return to the browser we must have a valid, non-expired URL-Content. It contains:

Block-Header – A well-defined 64-bit header.

U32 – Expiration time in seconds.

U8[20] – The name of the resource.

U8[20] – The SHA of a URL that contains this resource, w/o the host.

U8[?] – Meta-information, which includes mime-type, and maybe more.

The expiration time is the number of seconds that this URL will surly contain the specified resource. The time is from the time that this structure was downloaded from the Internet, until the time that we issue a request in order to get the file. A value of zero, for example, means that you may start download the file immediately, or revokes this structure.

There is a good reason why to use relative expiration time instead of fixed time. This gives the host server the option to say that this URL is valid for 5 minutes. Than, after 5 minutes the client can check the **active** URL. If this URL is unchanged, it extends this file for another 5 minutes. This is more flexible, and bypasses all the time-zones stuff.

The name of the resource is the SHA digest of it, as always.

The SHA of the URL allows us to attach it to a specific URL. This is more efficient than to put the URL itself. When implementing we will need to define it exactly (host, encoding, spaces, uppercase/lowercase, UTF-8, and so).

The meta-information contains extra information about the resource. :TBD: **Edan, Nadav, Yuval**: We need to check and see what is the best method for it. We can use a more strict method, which contains an integer that will specify a mime-type from a pre-defined table of supported mime-types. A more robust way will be to state that this is a variable-size filed will contain HTTP headers. This headers will be passed as-

is, without processing, to the user agent (browser). I think that I prefer the s[redacted] method, even that it's more expansive.

8.1.8. Resource-Download-Info

This structure contains the information needed to describe how to download a single resource. It works both with the protocols in the left column (with Content-Groups) and in the right column (without Content-Groups). It contains:

Block-Header – A well-defined 64-bit header.

U8[20] – The name of the resource.

U32 – The IP address for multicast or 0 if invalid.

U32 – The IP address for Reliable-UDP or 0 if invalid.

U32 – The IP address for HTTP transport or 0 if invalid.

If we need more information for implementing the protocols, we should add it to this structure. I am talking about RLC layers, about ports, about passwords, and so. This is why we have headers ☺.

8.1.9. Compression

Compression here is a process that maps any byte array to any other byte array, which is usually smaller. The reverse mapping called de-compression. The compression works on file basis, like ZIPs, unlike CABs.

The compression **MUST** be deterministic, and well defined. For any given uncompressed file, there is a single compressed file. It sounds trivial, but ZIP compressors, in general, do not do it! For example, you can tell the ZIP compressor to use normal compression or extra compression. Without being well defined, we will fail to implement the erasure code! This is very important, and should be compatible between versions of our product.

We can define that the first byte is the compression type, or zero for uncompressed resources. However, this still means that if one encoder decides that the compression method will be X, all other encoders must reach the same decision.

Since this is critical, as a safety guard, we shall add a checksum of the compressed resource in order to validate that this actually works. A simple 32-bit checksum should be enough; there is no need for a secure digest.

8.1.10. SHA Cache (for the client)

The cache, as seen from outside, is a **hash-table** of **compressed resources**. For each resource the cache holds:

1. The name of the resource (which is the key to the hash-table).
2. The U32 checksum of the compressed resource.
3. The compressed resource itself.

It is important to note that the cache does not hold the original URL, mime-type, extension, date, expiration, and so. This spec will never be coherent if it does. The cache holds some internal information for the LRU algorithm, or whatever we may implement, but there are no set and get methods for them!

:TBD: **Yossi** told me that from reading the cache article which **Yuval** has sent, he thinks we should add to the above list some values which indicate the cost associated with getting the resource. Then we will be able to implement some LRV algorithm. This is external information provided by the transport when the resource arrived. I didn't have time to read it yet.

The resource-level **operations** on the cache are:

1. Get the uncompressed resource data by its name – as stream to socket.
2. Get blocks from the compressed resource data by its name – for the decoder.
3. Get the checksum of the compressed resource by its name – for the decoder.
4. Store a (new) resource; Fill all the fields, and validate that the digest of the (uncompressed) resource and the checksum of the compressed resource match their expected values.
5. Put a listener that waits for a given resource to appear and then sends a message to some thread.

We need to store the compressed form in order to retrieve it when needed for the erasure code decoding. This saves the need to put the compression code in the client.

We might need to implement a second uncompressed cache if the client asks for a resource too many times, and we do not want to decompress it each time. My guess is that we can implement the decompression on the fly when writing to the output socket.

Other operations are for management only. Examples are create, destroy, compact, empty, set max allocated size, and so. There are no operations like enumerate, invalidate, update, remove, and so. We do not need them ☺.

8.1.11. Content-Group-Header

A Content-Group is a set of compressed resources. The Content-Group-Header is a single file, which defines the resources inside a Content-Group. It has two versions, which have minor differences between them. The first version is for resources (each entry describes a resource). The second version is for Content-Group-Headers (each entry describes a Content-Group-Header inside this Master-Content-Group-Header).

The file format is:

Block-Header – A well-defined 64-bit header.

U32 – The number of packets in the encoded file.

U16 – The erasure code block size that this file was designed for (512 bytes?).

U16 – Number of resources described in this file.

For each resource:

U32 – The offset of the compressed resource.

U32 – The size of the compressed resource (without padding zeros).

U32 – Checksum of the compressed resource. For compression validation.

(contains one of: for resources:)

URL-Content – Describes the content of the requested URL.

(and for Master-Content-Group:)

Resource-Download-Info – For downloading the Content-Group itself.

Each Content-Group-Header has a name, which, as always, is defined to be the SHA digest of it. Please note that since the header contains the SHA of the resources themselves, this name also defines the whole Content-Group. One cannot create two valid Content-Group-Header names with different content.

In the last version I changed the sizes to be a start offset and length. This was done in order to support that a single Content-Group will contain a resource that is access through several URLs. This is needed in order to support stupid HTML designers, which creates a single page that contains several resources with the same value. If we detect such a case, we can point both URLs to the same resource ☺.

:TBD: We may need to include some kind of priority bit for the first resource in this Content-Group. If this bit is set, it means that asking for this resource will cause, with high enough probability, requests for all other resources in this Content-Group. This means that if this first resource is required, the client should prefer this Content-Group above any other.

:TBD: Yossi has an idea about adding information about other Content-Groups that the user may want at this stage. I can add it to the client. However, the implementation of this kind of pre-fetching in the server is very complex. It will need a higher level of statistics (Markovian instead of zero order). We will actually need to "track" our users for it. I think that it's out-of-scope for the initial version.

8.1.12. Content-Group-Data

A Content-Group-Data is a single file, which is the Content-Group-Header, joined with the compressed resources and zero bytes padding, as defined by the header. The name of the Content-Group-Header defines the content group uniquely.

The Content-Group-Data exists only in the server. The client never gets this file as-is. For this reason, there is no need to give it a name depending on this file SHA digest. Using the Content-Group-Header name is sufficient.

8.1.13. Validation of Content-Group

Validation of Content-Group is a command to the cache, which gets a Content-Group-Name, and returns a Boolean, which is true if the Content-Group was validated correctly.

All the definitions in this paper leads to one thing: To create a **bullet-proof** mechanisms that will let the client know, for a 100%, if the return value should be true or false. And if it's false, this process can detect, with 100% accuracy in both directions, which files the cache already has. Later, we can consume bandwidth only in the sum of the size of these files. This is a case when technology can solve problems before they are even created ☺.

When a client gets a Content-Group name, it can search the cache for an entry with the given name. If it finds it, it can validate the digest of it. Then, it can access all the relevant files in the cache, and validate their digest too. When validating these files, it can detect errors that the simple checksum missed, and throw some kind of internal error exception for compression errors. If all files exist, the receiver can, in principle, create the Content-Group-Data itself.

8.1.14. Encoded-Content-Group

This file, if needed, contains a small header, and then a large number of data packets. It exists only in the server side. We need this file format if we want that the UDP server will have a simple job at runtime, something like a data carousel. :TBD: **Nadav** should help me define the exact parameters inside it.

As far as I can tell, each packet of data will contain at least these fields:

1. The encoded data itself, as defined in the block size above.
2. A **U32**, which is the seed to the random-bits generator that created the code of this block. This will allow us to send a small number of bits and describe a whole row in the code matrix.
3. A **U32** checksum of the Content-Group-Header name, the seed, and the data. The receiver can check that this block received correctly.

The last checksum allows us to detect transmission errors in this packet, since the UDP, as far as I know, does not have a CRC field. Also, and more important, it can detect misconceptions about which stream this packet belongs to. This is important, because the current suggestion of the implementation of the multicast in the Internet is lousy, and might cause one client to accept packets that were used by another group.

8.1.15. Master-Content-Group

Each host has, at any given time, a single active Master-Content-Group, described by the **ACTIVE** URL. It is almost a "normal" Content-Group itself. There are some differences, however.

The first difference, which is actually an addition, is that the resources inside it are actually Content-Group-Headers. This means that when the Master-Content-Group is downloaded, and it's downloaded differentially, the client has the most important Content-Group-Headers that this site serves. They are actually "pushed" to the client, using multicast (or unicast / HTTP transport), so that each client will not ask for them independently.

The second difference is that the FEC we use in order to send the Content-Groups-Headers is the FEC with priority-based information. This gives the client access to the most important Content-Groups-Headers first, and to the less important (from the most important Content-Groups) a little later.

The third difference is that this header is little different. Instead of the URL-Content, which describes a URL, this header contains the required information for downloading the Content-Group itself, when needed. This information contains 3 IP addresses that the client will use for downloading the Content-Group using one of the left-column protocols.

We need these IP addresses here, and not inside the Content-Group-Headers themselves, because they are dynamically allocated and de-allocated. So, if a server decides that a Content-Group should move from multicast IP group X to Y, we will not need to download the Content-Group-Header again.

Each Master-Content-Group has a name, which is its SHA digest. It is important to understand that this name defines uniquely, recursively, all the accelerated information that the host wants to push to the client at this moment! In order to get this information, however, there are several levels of indirection, which actually allows do download the information differentially and on-demand.

Nadav, Yuval, and Omer: Please read this:

Another important note is that this push idea is **optional**. The server can use it for a lot of Content-Groups, for few, or not use it at all. If the client does not have a Content-Group-Header, it will connect to the server, and the server will send it to the client. It is only a way that the server will push the most important Content-Group-Headers to the client. This is actually some kind of pre-fetching. **So, the famous 100k problem can be 20k problem using the priority-FEC, and can disappeared at all if the site administrator do not want this kind of pushy pre-fetching.**

8.1.16. The ACTIVE URL

The **ACTIVE** URL is the root of all the information in a host. The client must obtain it in order to get even a single bit of accelerated information. The client never stores this block in the cache. It keeps it in-memory, until it expires.

It is a block in this format:

Block-Header – A well-defined 64-bit header.

U32 – Expiration in seconds.

U32 – The IP of Reliable-UDP server for this host, or 0 if not available.

Resource-Download-Info – For the Master-Content-Group-Header.

Resource-Download-Info – For the Master-Content-Group itself (SHA ignored).

The expiration field contains the number of seconds that this block is alive from the time the client got this block until it expires. The client must discard this block after this amount of time.

The IP for Reliable-UDP is the IP of the server that is used in order to get resources that were not pushed by the server. A value of 0 means that only pushy multicast is available.

The Resource-Download-Info for the Master-Content-Group-Header contains the SHA of the active Master-Content-Group and the IP addresses in order to get it from. Using this information, the client can download this header using any of the 3 protocols in the right column.

The Resource-Download-Info for the Master-Content-Group contains the IP addresses in order to download the Master-Content-Group itself. The SHA field inside must match the SHA of the header (duplicate; ignored). The client can download it differentially, using the priority-information FEC, using any of the protocols in the left column.

8.1.17. Locating Content-Group by resource

Even when having all the Content-Group-Headers, and the cache full of information, this is not as simple as it sounds. First, the client still has to give some meta-information on the file, like it's mime-type. Without the mime-type the browser will not work. Please notice that the mime-type is **not** a part of the file. This means, for example, that two sites may use the same file with different mime-type to cause different plug-ins in the browser to process the file. It is preferred that we support this feature, because it can save us from getting into trouble in the future.

The client needs to know the required URL, and use it in order to get this information:

1. Which is the current Master-Content-Group.
2. What's the SHA of the file?
3. There is no need to get the mime-type from the URL itself.

Then, the client need to search the Master-Content-Group for files with the given SHA digest. If it doesn't find any, it should fallback to the compound method defined in the protocol section. This means that the resource is not included in the hottest pages, or that it does, but the client didn't have enough time to know about it.

If it finds it exactly in one Content-Group, it should download this group. This should not be overkill, since the Content-Groups is designed for it. For example, let's assume that user want to see the page **x.html**, which is not accelerated, but embeds the accelerated images **1.jpg** and **2.jpg**. We can decide to put both images into a single Content-Group. Asking for **1.jpg** will cause **2.jpg** to be downloaded too, but it's OK since we believe that the browser will ask for it eventually. We get pre-fetching for free ☺.

We might, however, find more than one Content-Group that embeds the file with the given name (SHA digest). One case is when this is the first file in the Content-Group, and it marked using the priority bit. This means that after returning the data of this resource, the browser will surely ask for all others in this Content-Group, and it is better to download this Content-Group right now.

Another case is when this file is a normal resource in more than one Content-Group ☹. This should not happen a lot. In this case we can:

1. Pick one group by random.
2. Pick the smallest group.
3. Pick the group with the best protocol.
4. Pick the group which most of it is in the cache.
5. Wait for a clue from the browser.

The wait-for-a-clue method is the harder to implement, but will give the best results. We do not have to implement it in version 1 of the client. In this method, we should wait for a while, for the browser to ask for another resource. For example, the image **1.jpg** exists in several Content-Groups, and we cannot decide which one to chose. But, after 10ms, the browser asks for **2.jpg** using a concurrent TCP connection in a different thread. At this stage, we find that there is only one Content-Group with these two files, and it does not contain extra files. We should pick it.

This is hard to implement. There is a huge difference between something that I will call a "transactional" multi-threaded program than this one. Normal servers look to the outside world as-if they work in transactions. When they are two concurrent connections, the server can be viewed as if it does one of them first and the second one later. Here, **this is not the case**. Getting query for only **1.jpg** would cause the client to download **CG1**. Getting query for only **2.jpg** would cause the client to download **CG2**. But getting both caused the client to download **CG3**.

Except the protocols themselves, this is the only place in the client that we need heuristic, multi-threaded algorithms. It is needed only if the main HTML is not accelerated. If the main page is accelerated, we will not get to this kind of decisions too much.

8.1.18. URL to SHA conversion

In order that the Content-Groups will work, the client needs to know the SHA of the resources inside URLs. This is a dirty task; it creates all the dirty problems of coherency we are trying to avoid ☹.

I though about several methods, but most of them have major flaws. The best method was that the name of the URL would be its SHA. This method is valid for resources, but not for HTMLs, because we do not want that the user will see it. We didn't like it, because we do not want to be intrusive. Another intrusive method suggested by Yossi was to add a TAG to the HTML source which will describes the SHA of it's resources, and that the browser will ignore it. This method is both intrusive and CPU intensive.

A non-intrusive method is to ask the server about the SHA of a resource. We will use it in the combined protocol, when the client asks the server to send it a URL, and the server sends a packet with the description of the resource, which contains it's SHA. The client may abort the download if it has the resource in its cache.

Although the above method works, and we shall use it, it is not suitable for downloading of Content-Groups. We need something that contains lots of URLs and their SHA digests. This is why I want to use the URL-Content object.

9. Unsolved problems

:TBD: Very fast network support.

:TBD: Maximal Content Group size.

:TBD: Paid pages.

Reliable Unicast FEC-Based Protocol Design

Edan Ayal,
R&D Department,
BandWiz Ltd.

Table of contents

1. Introduction

This paper contains the description of BandWiz reliable unicast FEC-based protocol. In general, the goal of this protocol is to provide a reliable, highly efficient, TCP-friendly unicast transport layer to be used in the UltimaCast system. This protocol would be the best alternative in absence/termination of multicast coverage as long as there're no network access problems.

The motivation of replacing the reliable unicast TCP protocol is the shortcomings of basic TCP. These would be (1) connection establishment and teardown overhead (2) poor performance when packet loss increases (3) inability to separate loss detection and retransmission. Some of these issues were addressed in late extensions to the basic TCP, but these extensions are rarely deployed. Our protocol would utilize the existing UDP infrastructure rather than the TCP.

The uniqueness of this protocol is the usage of pseudo-infinite FEC. Pseudo-infinite FEC ("FEC") enables us not ever to retransmit a lost packet. Instead, some other FEC packet is generated by the engine and used. So, basically we never need to send negative acknowledgements (AKA Nack) – just to report on global success or some fatal error (and some periodic reports as will be shown later). This makes our protocol closer by nature to Real-Time semi-reliable protocols, like VOIP protocols as RTP, rather than to reliable protocols like TCP. This should make the transport protocol itself much simpler.

2. Concept

The basic concept consists of a client and a server. The client wishes to receive a known reliable-unicast accelerated resource from the server. The client initiates a request, and the server responds. The session terminates on success or on fatal error.

There are two different operation modes. In the first operation mode the client has no previous useful information as to the resource about to be requested. The second operation mode takes place when the client has some useful information regarding the resource, and the protocol wishes to utilize this information in order to achieve faster (differential) retrieval of the resource. I shall describe these operation modes later.

For simplification, we limit the protocol to be "one-way" – that is, for each given session, there's a well-defined client, which wishes to retrieve a resource from a well-defined server.

3. Additional Requirements and Design Implications

3.1. Network Requirements

3.1.1. TCP Friendliness

By TCP-friendliness we mean congestion control. A network protocol, which does not perform congestion control, threatens unfairness to competing TCP traffic and possible congestion collapse. Is it also referred to as "Inter-protocol Fairness".

Actually, the TCP-friendliness imposes a stricter requirement. Any existing TCP connection, which shares the physical layer with us – its performance should not improve if we replace our connection with a TCP connection.

In order to achieve this the sender must dynamically adopt its sending rate so that the resulting packet loss, which is the indication to congestion, is decreased. The receiver should provide the sender with the appropriate information so that it would be capable of avoiding congestion.

Any congestion control scheme needs continuous reports from the receiver as regards to the packet loss statistics that it has encountered. So, we would need to add these periodic reports and control their timing and rate at the receiver side.

3.2. General Requirements

3.2.1. Concurrent Operation

The receiver and the sender should be able to handle multiple concurrent requests. That implies multithreaded operation of all components (e.g. FEC CODEC).

3.2.2. Resource Bounding

Both the receiver and the sender need to monitor and control their system resources.

3.2.3. Minimize Server Resource Consumption

Minimize the server resource consumption by moving all the possible activities to the client.

4. Protocol Detailed Description

4.1. General

The protocol is built as a variant of RTP (Real-Time Protocol). This donates the framework. However, RTP intentionally does not provide a congestion control scheme. Therefore, I've adopted a TCP-friendly congestion control scheme described at [3] and [4].

As stated before, there are two different operation modes. The first operation mode takes place when the client has no previous useful information as to the resource about to be requested. The second operation mode takes place when the client has some useful information regarding the resource, and the protocol wishes to utilize this information in order to achieve faster retrieval of the resource.

4.2. Basic Description

Regardless of the operation mode taken, at the IP network layer, the protocol uses the scheme described below. As stated before, the receiver periodically sends reports back to the sender. The sender reacts by tuning its sending rate:

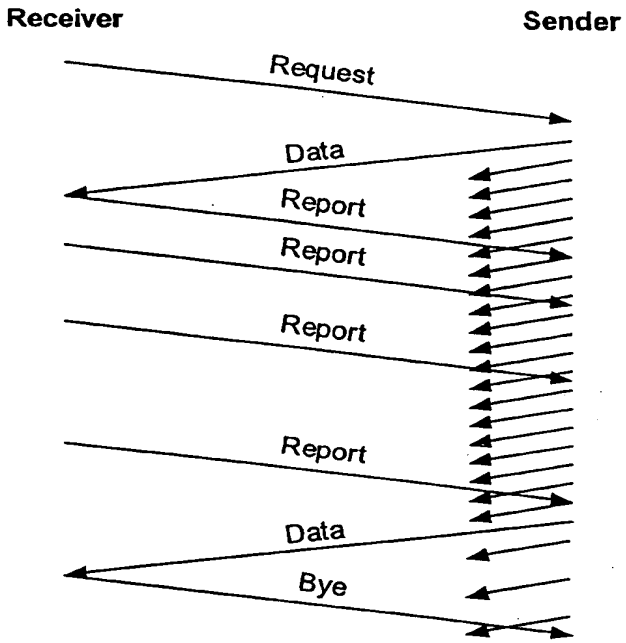


Figure 1: IP network layer view of the protocol

4.3. Block Diagram

The following figure depicts the basic block diagram of the FEC-based reliable unicast transport. The following two sections would explain the different operation modes and the role of each component within it.

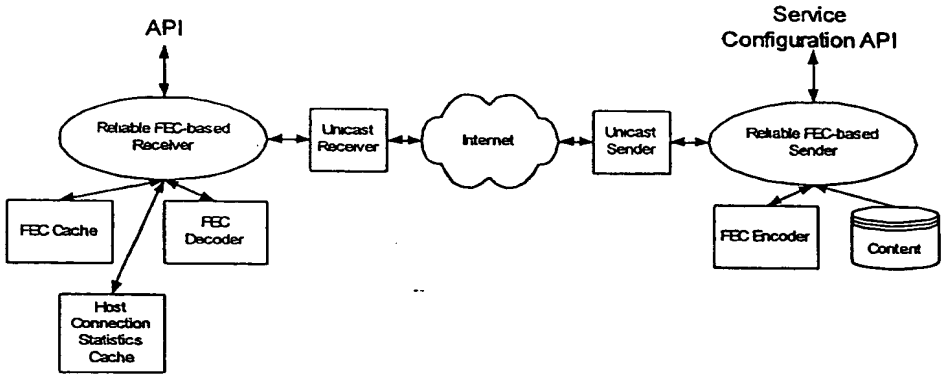


Figure 2: Basic block diagram

4.4. Operation Modes

4.4.1. "No Previous Useful Information" operation mode

The "Reliable FEC-Based Receiver" (RFR) accepts an external invocation asking it to fetch a certain resource from a certain server.

The RFR searches the "Host Connection Statistics Cache" or a hint about the characteristics of the previous connection with that server. The RFR asks the "Unicast Receiver" to initiate a request, using the characteristics found (default, if not) within the host cache.

The request is accepted at the server, and the server starts a continuous operation of pumping encoded FEC packets towards the client, according to the initial rate specified by the receiver.

Every received valid FEC packet is inserted into the FEC cache, and the FEC decoder finds out if the whole content has been received. If so, the "Unicast Receiver" is kindly requested to close the session. When the sender received the "close" command, only then does it halt the FEC generator.

During the data reception the "Unicast Receiver" periodically sends reception statistics report packets to the "Unicast Sender". The report packets are used by the server to tune the send rate: avoid congestion while transferring as much data as possible. The rate of these report packets themselves is also tuned by the receiver according to various computations.

4.4.2. "Existing Previous Useful Information" operation mode

This operation mode differs from the basic one by the fact that the client has some previous useful knowledge regarding the content. Therefore, it can optimize the retrieval process by asking for completion data. The sender does not send just FEC packets, but XOR of three random FEC packets.

This change does not influence the Unicast components – except the needed change in the initial request.

Like before, the FEC decoder checks for completion on every received packet, but the check algorithm is different.

The FEC cache is also a bit different than the previous FEC cache, but they could probably be united into the same cache somehow.

4.5. State Diagrams

The following figures describe the state diagrams of the receiver and the sender service. Please note that the detailed timing issues (protocol clock, determination of sending rate, RTT and the various timeouts) are not described here but in the following section. Note that the values of the various timers are dynamically changed.

4.5.1. Receiver

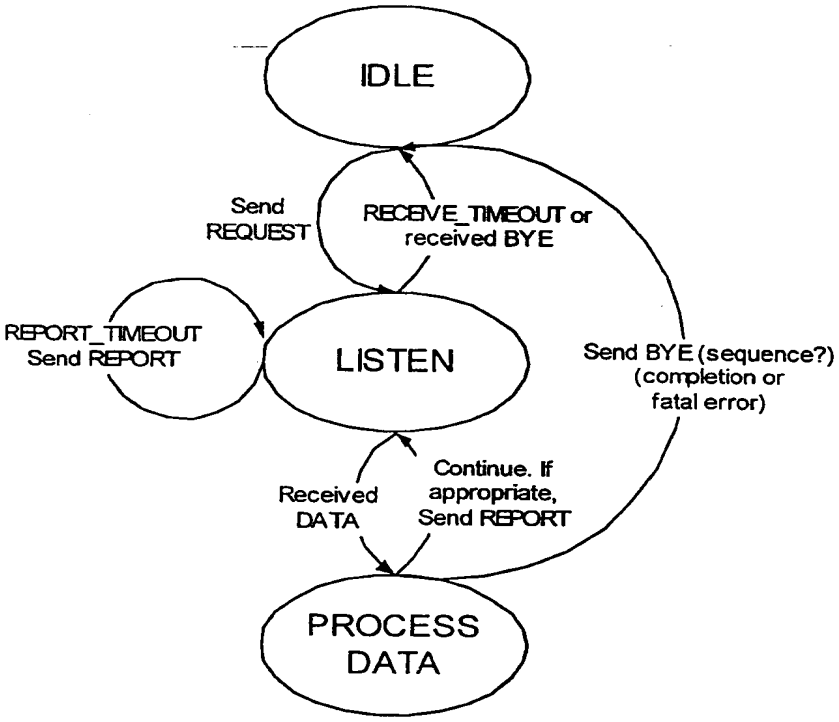


Figure 3: Receiver state machine

The time values that appear in this state diagram are:

- Receive timeout: For both the initial response (first data packet) and the following data packets. The timeout on general data reception during the session is clear. Regarding the initial response: After sending the request, the client waits for this period of time for the server to reply. The application can

configure the protocol layer to resend the request and give up only after Y rounds. The default is just a single round. Measured from the time of the request, and once data packets are being received – from the latest data packet.

- Report timeout: the receiver sends a report packet based upon two different mechanisms: First, the report is sent when X new data packets have been received, or if this timer elapses. The goal of this timer is to supply important reports when data packets are not arriving fast enough. Measured from the time of the initial request, or the time of the latest report sent.

4.5.2. Sender

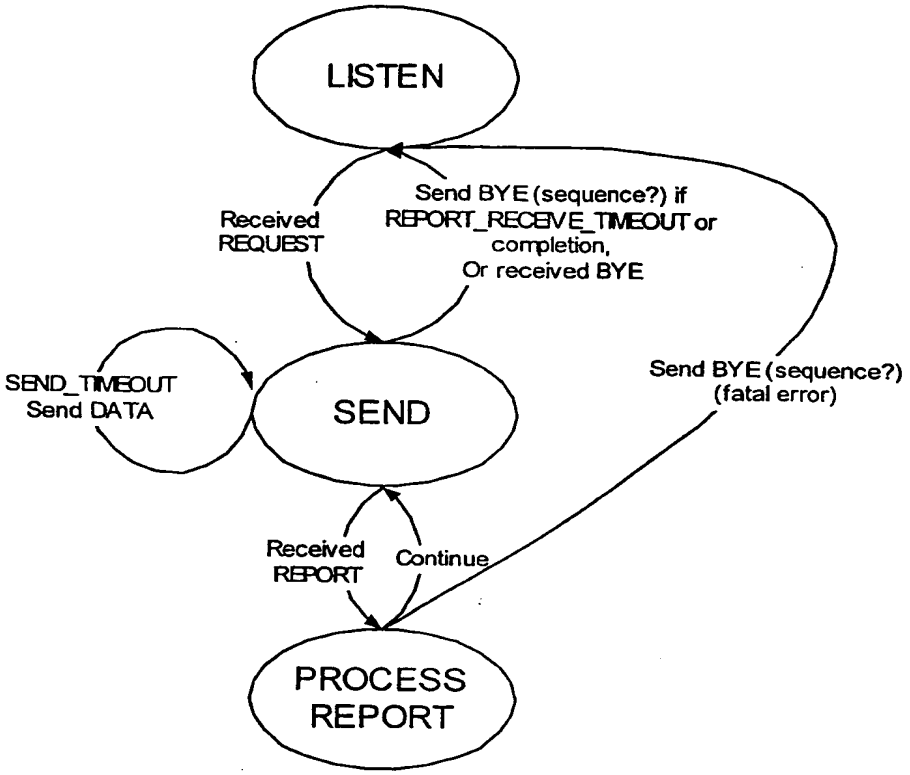


Figure 4: Sender state machine

The time values that appear in this state diagram are:

- Send timeout: The "wait" period between sending single data packets. The actual rate of transmission. Dynamically changed as a result of the report processing to reflect the available bandwidth of the channel. Measured from the latest data packet sent.

- Report Receive timeout: measured from the latest report or the initial request (if none), this timer reflects the need of the server to receive continuous congestion control signals.

4.6. Timing

4.6.1. Protocol Clock (Decision Cycle) – RTT Estimation

Both peers are constantly performing dynamic modifications of basic variables. According to both [3] and [4], the recommended decision frequency should be no more than once per RTT – the Round Trip Time - for the relevant path. This will prevent the application from over-reacting in times of congestion.

The RTT is a moving target, so we need to constantly reevaluate it.
The TCP RFC [1] suggests an estimate to RTT, called SRTT (Smoothed RTT):

$$SRTT = (\alpha \cdot SRTT) + ((1-\alpha) \cdot RTT)$$

Where α is the smoothing factor, around 0.8-0.9. (is this the "exponential moving average"???)

In our system, the SRTT should be estimated by the sender according to the reports received from the client. Then, it should be somehow conveyed, when appropriate, to the receiver so that it could also adjust its protocol clock and other timing variables accordingly.

We still need to determine the protocol clock. New input arrives only with a report packet. A report packet would cause the protocol to perform another cycle if the time from the previous cycle exceeds SRTT.

The implication of bad RTT synchronization between the peers is disastrous, so the report packet should include the current RTT for validation when it receives such value. Additionally, on the decision cycle, the RTT should be delivered redundantly to the receiver to cover possible packet loss. In the first implementation of the protocol the RTT would be inserted in 2 successive data packets. If the sender still does not receive such acknowledgement, it should resend it and be tolerant to bad-synchronization-related problems.

4.6.2. Report Rate

How often should the sender reevaluate the SRTT itself? New data is available only when new receiver report arrives, so this question actually is: how often should the receiver send reports? The tradeoff here is between accurate continuous estimation of SRTT (for proper congestion control as well as high bandwidth utilization) and bandwidth consumption (frequent reports would contribute to bigger congestion...).

RTP [1] suggests that the report traffic would occupy a constant (small) fraction of the bandwidth – 5% is recommended. If we take into consideration that in our framework data senders do not send report packets as in RTP, where ¼ of the 5% is allocated for that, we get 3¾%. However, in RTP (actually RTCP) the report channel has additional functionality so we can settle for a bit less. For the initial implementation, the percentage would be set to 1-2%. E.g., for report packets of size 50 bytes, data packets of size 500 bytes and report percentage of 2%, a report should be sent every 5 data packets (10 if 1%).

Reducing the report rate according to the stability (variance, in percentage) of the bandwidth seems logical and would be further investigated in future. However, reducing the rate for high bandwidths is wrong since an accurate estimation of RTT is extremely important, and sudden changes (for the good or for the bad) should be reflected as quickly as possible regardless of the connection speed.

It is important to note that the client report should normally be triggered by a received data packet and not by some timer expiration - for accurate RTT estimations. The following mechanism is suggested: The rough timing to send the next report packet is set according to the latest reported bandwidth - e.g. after X more new data packets or after T timeout. Once the client receives the expected data packet, the report packet is sent. If no proper data packet arrives, the report would be sent on timer expiration, and it should include a flag to specify that the RTT estimator at the sender should ignore it. The value of this timer is the estimated arrival time of the X*1.25 packet. E.g. if report is sent every 4 packets, and the 4th hasn't arrived yet, the report should be sent in the worst scenario on the estimated arrival time for the 5th packet.

The very first report packet sh[redacted]be sent immediately after the first data p[redacted] is received. To allow proper behavior even when this packet is lost, a report packet should be sent right after the 3rd and 5th data packets as well, regardless of the bandwidth.

4.6.3. Bandwidth Estimation

The current bandwidth can be computed from the report packets in the following way:

$$Bandwidth_t = \frac{data\_packets\_received_{t-1\,to\,t} \cdot data\_packet\_size + report\_packet\_size}{NTP_t - NTP_{t-1}}$$

$$data\_packets\_received_{t-1\,to\,t} = [(EHSNR_t - CNPL_t) - (EHSNR_{t-1} - CNPL_{t-1})]$$

Where EHSNR stands for "Extended Highest Sequence Number Received" and CNPL for "Cumulative Number of Packets Lost".

Especially when the number of packets is relatively small (<10 packets), it makes sense to smooth the bandwidth estimation by several ways. First, in a similar way to the RTT smoothing. Second, by computing the bandwidth over a wider window – that is, from "i-n" (instead of "i-1") to "i". Note that in such case, the "report_packet_size" factor should be multiplied by n.

Every decision cycle the system estimates the current (smoothed) bandwidth, and conveys the information to the receiver for report rate tuning. Like before, this data should be sent redundantly to cover possible packet loss, and should be validated by including it within reports. Once again, if such validation is not present, the sender should resend and be tolerant to bad-synchronization-related problems.

4.6.4. Sending Rate - Congestion Control

TCP implementations use an AIMD (Additive Increase Multiplicative Decrease) algorithm [4], where the sending rate is gradually increased when loss rate is absent, but instantly halved when loss is encountered. This rapid backoff is crucial to avoid congestive collapse and TCP friendliness, and should be kept. Currently I present an additive increase like TCP. In future some other increase mechanisms should be tested (like exponential increase).

The parameter being modified every decision cycle to reflect the current sending rate is the "Inter-Packet-Gap" (IPG, [3]):

$$SendRate_i = \frac{data\_packet\_size}{inter\_packet\_gap_i}$$

Whenever loss is reported, it is halved:

$$SendRate_{i+1} = \beta \cdot SendRate_i$$
$$\beta = \tfrac{1}{2}$$
$$\text{inter\_packet\_gap}_{i+1} = \frac{inter\_packet\_gap_i}{\beta}$$

(Duplicate packets are always discarded and regarded as indication to congestion).
In the absence of packet loss, the rate is additively increased:

$$SendRate_{i+1} = SendRate_i + \alpha$$
$$\alpha = \frac{data\_packet\_size}{C}$$
$$\text{inter\_packet\_gap}_{i+1} = \frac{inter\_packet\_gap_i \cdot C}{inter\_packet\_gap_i + C}$$

Where C is a constant with the dimensions of time.
If the decision cycle is SRTT and C is set to SRTT, we get (again, [3]) a similar behavior (slope) to TCP. In order to maintain the slope when the decision cycle is different than SRTT, we should keep the product of the decision cycle and C as $SRTT^2$. So, once the decision cycle is known, we set C to the proper value and compute the needed change in the inter packet gap.

$$C \cdot CurrentDecisionCycle = SRTT^2$$

4.6.5. Receive Timeout

This timeout is used by the receiver to detect fatal disconnection errors. The timer is started on the initial request and restarted after each received data packet. The initial value is predetermined (say 15 seconds, during which the request should be resent 2 more times). The other values should be dynamically modified to reflect the speed of

the connection. If no data packets are received for X report cycles, the protocol should report a fatal error to its API. In the initial implementation I set X to be 3.

4.6.6. Report Receive Timeout

This timeout is used by the sender to detect fatal disconnection errors. The timer is started on the request and restarted after each report packet received. The initial value is predetermined (say 15 seconds). The other values are not and should be dynamically modified to reflect the speed of the connection. If no report packets are received within X decision cycles, the protocol should report a fatal error and abort. In the initial implementation I set X to be 3.

4.7. Packet Format

4.7.1. General

Some of the mandatory fields in the RTP format are redundant in our framework. We can either implement a customized, RTP-like format and save the extra bandwidth, or we can comply exactly with the RTP standard and lose some bandwidth. Complying to RTP might have some good implication as this standard is a familiar one and might have special treatment at routers and firewalls – e.g. RTP header compression done between two CISCO routers (AKA CRTP – Compressed RTP). My decision is to have customized yet very RTP-like packet format, so that if we encounter in future the need to comply completely with RTP, the changes would be easy.

The following section describes the various packet types that participate in our protocol.

4.7.2. Request packet

Used by the client to initiate the session.

Field	*Size*	*Comments*
BandWiz Version	4 bit	0001
Packet Type (PT)	8 bit	REQUEST or COMPLEMENTARY_REQUEST
Timestamp	32 bit	Duplication guard. No need for accurate time as a report packet is sent when the first data packet arrives. (If it is really 2 separate requests that come from the same host - should be coordinated by the proxy to request once, and they would probably wont have the same timestamp)
Resource Identifier Length	16 bit	
Resource Identifier	"Resource Identifier Length" bit	SHA...???
Flags	8 bit	
Includes Start	1 bit	

	Offset		
	Includes Length	1 bit	
	Unused	6 bit	
Start Offset		0 - 64 bit	In bytes. 0 for beginning. Supports 4Gbyte.
Length		0 - 64 bit	In bytes. -1 for EOF (end of file). Supports 4Gbyte.

4.7.3. Data packet

Used by the sender to transmit the content.

Like in RTP, data length is not given, and is the reminder of the packet after the header is read.

Field		*Size*	*Comments*
BandWiz Version		4 bit	0001
Payload Type		8 bit	TBD. 200-204 cannot be used.
Sequence Number		16 bits	FEC block number. If A XOR B XOR C – should be in data header (???)
Timestamp		32 bit	
Flags		8 bit	
	Includes number of required FEC packets	1 bit	
	Includes RTT	1 bit	
	Includes Bandwidth	1 bit	
	Unused	5 bit	
Number of required FEC packets		0 or 16 bit	
Estimated RTT		0 or 32 bit	In milliseconds. Support the range from 1ms to ~65 sec.
Estimated Bandwidth		0 or 64 bit	In bytes per second.
Data		Rest of packet	If A XOR B XOR C, data header should include A, B and C (???)

4.7.4. Report packet

Used by the receiver to report the status to the sender.

Field	*Size*	*Comments*
BandWiz Version	4 bit	0001
Packet Type (PT)	8 bit	REPORT
NTP Timestamp	64 bit	Report send time. Accurate because used for RTT estimation.
Fraction Lost	8 bit	From previous report. Do we need this???
Cumulative Number of Packets Lost	24 bit	
Extended Highest Sequence Number Received	32 bit	Low 16 bit are the highest sequence number received. High 16 bit denote the cycle.
Interarrival Jitter	32 bit	Do we need this???
Flags	8 bit	
Includes RTT	1 bit	
Includes Bandwidth	1 bit	
Unused	6 bit	
Latest RTT	0 or 32 bit	As received from the sender
Latest Bandwidth	0 or 64 bit	As received from the sender

4.7.5. Bye packet

Used by both the receiver and the sender to indicate session termination.

Field	*Size*	*Comments*
BandWiz Version	4 bit	0001
Packet Type (PT)	8 bit	BYE
Reason Code	8 bit	TBD
Reason Length	8 bit	String length
Reason String	"Reason Length" bits	The reason code should be piggybacked. Also, BandWiz Version.

5. Software Architecture

TBD.

5.1. Software Interface

TBD.

6. Future Improvements

- Streaming: transmit the resource in chunks to enable gradual rendering.
- [Doron] Stateless simplified server: Sessions do not exist. The client packet reports become a new request to the memory-less server, specifying the resource, required range and format as well as the available rate. The client asks for small parts (in the report rate) and tunes the rate according to the loss rate it encounters. For load balancing: the requests could be addressed to different servers...
- [Doron, Yuval] Moving as much computation as possible to the client. I cannot figure out how to compute the RTT at the client side and apply it to the session without an additional report to the sender (or unacceptable whole decision cycle delay).
- Stable connection – decrease report rate: On stable connections the report rate can be carefully decreased.
- AIMD: A more aggressive greedy increase as well as a less radical backoff on minor loss detection. Not TCP-friendly, but better bandwidth.

7. References

1. RFC 793: TCP. http://www.faqs.org/rfcs/rfc793.html.
2. RFC 1889 (modifications draft to...): RTP. http://www-mice.cs.ucl.ac.uk/multimedia/misc/avt/IETF46/internet-drafts/draft-ietf-avt-rtp-new-05.txt
3. TCP-Friendly Unicast Rate-Based Flow Control – J. Mahdavi & S. Floyd. http://www.psc.edu/networking/papers/tcp_friendly.html
4. RAP paper. http://nw.dongguk.ac.kr/resource/ClassResource/Seminar/sgna/RAP_rate_adaptation_protocol/RAP.ps
5. RFC 1323: TCP Extension for High Performance. http://stevep.ne.mediaone.net/doc/rfc/RFC/1323/rfc1323.txt

BandWiz

UltimaCast 4 WWW

System Design

Table of Contents

List of Figures

List of Tables

1. Abstract

This paper contains an in-depth description of **UltimaCast 4 WWW** system. This is "the" technical paper that we have. All the things that the system can and can't do are specified in this document. The implementation of the system will be based on this document.

This paper, among other things, contains a block-level description and interfaces between them. In the near future, even before this document is completed (surly before we agree on it as a whole), we will start to break it into pieces for inner-block design and testing.

I though about using Word's Version support, but it do not work. When having only 4 version, the file grew from 150K to 32M, a factor of 250! Instead, each version will have a name that contains the date I crated it. You can edit the file and send it back to me. The integration is simple: Word has a "Compare Documents" option, in which it will light all the changes, and allow me to accept them or reject them. This will work good unless I totally changed the document until you send me your comments. So, please send your comments quickly, and on the last version of this paper. Please, do not updated embedded Visio drawings, layouts, or move stuff from one place to another. If you have a better way – please tell me soon ☺.

The red, bold ":TBD:" means things that "To be discussed", implying that I do not have a good solution for them right now, or that I didn't have time to finish them.

2. Goals and Requirements

2.1. Commercial goals

2.1.1. Being non-intrusive

The system should be near-non-intrusive, or maybe, non-intrusive at all to the content

2.1.2. Hit counting

We should give our accelerated site an accurate hit counting and bandwidth. This might be that thing that we are paid by.

2.1.3. Point of presence

We want to be able to — client & servers in the Internet even before having multicast capabilities in the Internet.

2.1.4. Interoperability

We want to design a file format that can be used as-is in different computer architectures. We want robustness, version control, and some kind of backward and upward compatibility.

This file format should allow memory-mapped files too.

2.2. Performance goals

2.2.1. Communication Efficiency

It is common for us to say, "TCP is inefficient by 30% relative to UDP". We need to prove it! Anyway, we need to have higher throughput, and lower latency, while maintaining TCP-friendliness. We need to use pre-fetching and differential downloading in order to help us achieve these goals. In general, we need to give the user a smoother and cleaner browsing experience.

For example, the clients will send statistics update. This is OK, because even today when they access HTTP the server get a hit. But, unlike HTTP servers, I want to use unreliable UDP for this statistics. This means that this packet might never get to the server. I do not want to lose this information, so I will make these packets commulative, meaning that each packet contains the information on the previous ones. This means that we will need some kind of client ID. Because we do not want to track our users, we can use a random 160-bit id, per client, per server, per session. So, no one can tell that the same client who seen page X at site Y yesterday is the client which connects to page Z at site W. Of course, they can use the IP address, but we do not supply more tracking information!

2.3.2. Using cryptographic tools

When thinking about channel coding, the common intuition is that it's adds to the robustness of the system. Examples are parity codes, Hamming codes, Reed-Solomon codes, BCH codes, Trellis codes, Turbo codes, and so.

Our erasure code is different – it enhances errors, rather than reducing them. For example, if we use a simple checksum code, and fail to detect an error in a single packet, the whole system will collapse. This packet will trash the current resource, and all the files in the Content-Group. Then, it will trash every Content-Group that has a common file with this one. If, for example, each page has a menu on the side, and one image of this menu has a bad block – it will trash all text and data of **all the pages in this site**. This effect will never go away. The only way to fix it will be to uninstall and reinstall the UltimaCast 4 WWW system ☹.

In order to prevent it, we need a strong, secure code. This is why we should use SHA digests. If the URLs in the HTTP server will be left unchanged, we might need some kind of hash-table to map them into SHA digests. Failing to keep it up-to-date will cause displaying of older version of the text, or download the current version again, as happens today. But it will not have the magnifying effect that I described above.

3. System Overview

3.1. System Block Diagram

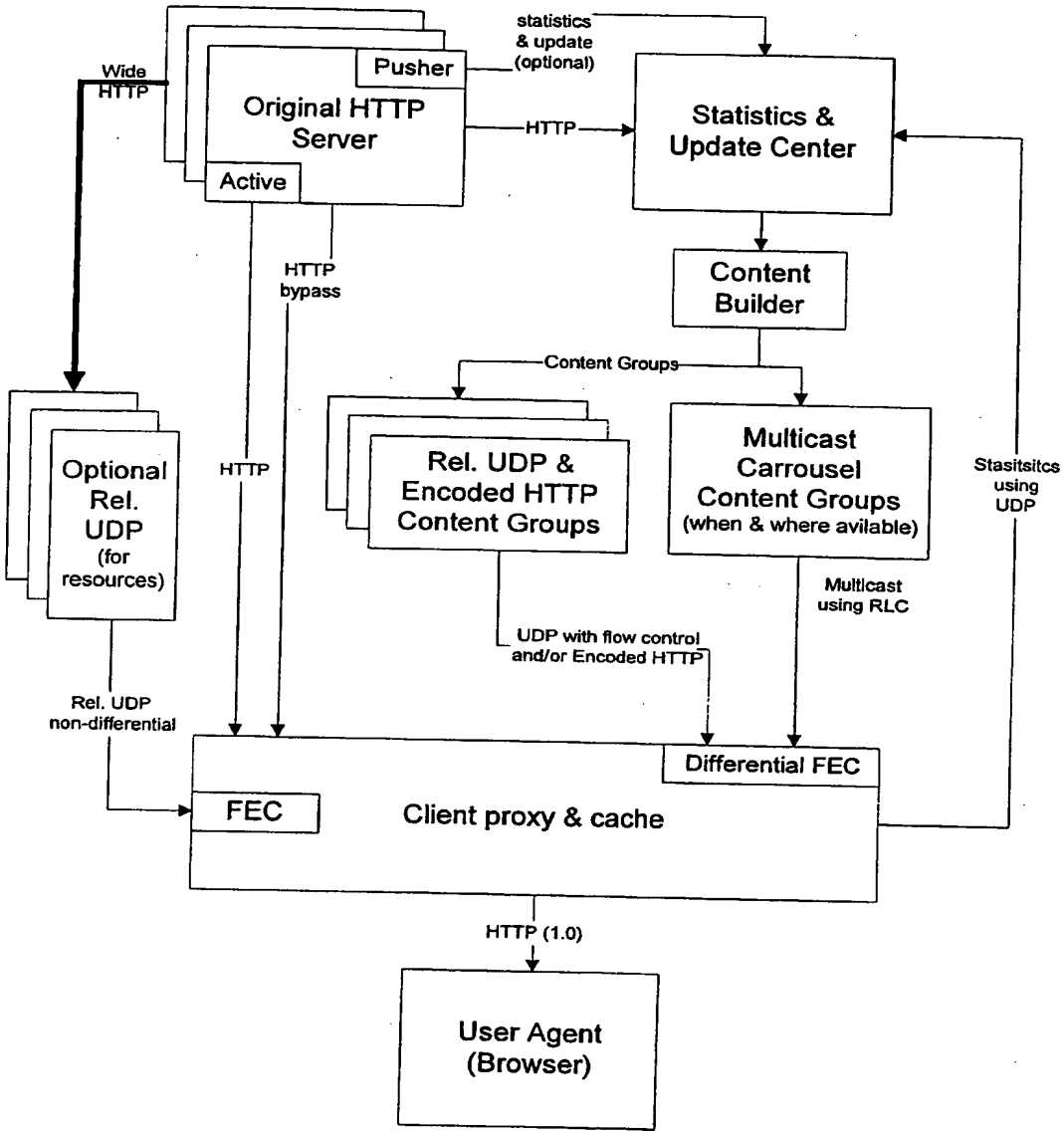


Figure 1 - System Block Diagram

3.2. Quick explanation

Each block is a system component, which can be in a different computer. The "3D" blocks are blocks that we might need a cluster ("farm") of servers in order to implement.

Each line is a connection between blocks. The connection protocol is written on the line itself.

Only the "Client proxy & cache" block is the "client". Other components are the "server". The client gets information from the server, and feed it back with statistics.

4. Common scenarios

4.1. Location of each component

:TBD: The minimal configuration of the server and client. The maximal configuration of the server and the client. Integration with the ISP.

4.2. Original HTTP site

:TBD: Modifies a lot. Dynamic HTTP. Static HTTP. And more.

4.3. User behavior

:TBD: Some common user scenarios. First time, after a month, after one hour, organization cache, cache in ISP, and more.

:TBD: I need to describe the initial connection scenario, and what we send and receive. Should this be the place?

5. Protocol detailed description

:TBD: Describe each of the lines in the drawing. I guess that the protocols for the client are more or less defined. I need to define, for example, the protocol of the statistics update from the client; or the protocol from and to the content builder.

6. Module detailed description

6.1. Original HTTP Server

We should be as non-intrusive as possible. All we want are three things:

1. A URL which will be the "root" for everything.
2. Statistics for the normal HTTP hits in the site.
3. Notifications about important pages that were updated.

We can do well without the last two. The first, however, is a little bit tricky. It is better to have it.

:TBD: **Edan, Omer**: We should think about Cookies. They will cause a big problem!

6.2. Statistics & Update Center

Will get information from these sources:

1. The statistics that the clients sends.
2. The statistics that the original server sends (optional).
3. Detect updates in the original site, by polling.
4. Notifications about updates in the original site (optional).
5. Administrator command.

Based on all this information, the statistics center will decide about building Content-Groups, and transmitting them over Multicast and/or Reliable-UDP and/or HTTP. Then, it will issue these commands to the Content Builder.

:TBD: Describe it a little more.

6.3. Content Builder

Builds a content for a given URL. This will recursively detect all the embedded resources inside a URL, and pack them in a Content-Group. This builder needs an HTML parser, compression utilities, and more. :TBD: **Yuval**: Should we buy there tools now?

Then, depending on the command from the statistics center, it will deploy these Content-Groups to the multicast carrousel and/or to the Reliable-UDP server cluster.

:TBD: Describe it a little more.

6.4. Multicast Carrousel Content Groups

This is a stupid data carrousel. It does not accept packets from anyone, except from the content builder. I think that a single PC will be able to serve all the hot pages in the entire site. We need the multicast to be deployed for this component.

:TBD: Describe it a little more.

6.5. Rel. UDP & Encoded HTTP Content-Groups

This is a server cluster for serving clients using reliable-UDP and Encoded HTTP Content-Groups.

:TBD: Describe it a little more.

6.6. Reliable UDP for resources

If we want to use the Reliable UDP for serving large resources, we need this component. It does not need to get statistics or anything like it. It should simply get requests for URLs and serve them. It should have a cache, however.

:TBD: Describe it a little more.

6.7. Client proxy & cache

This is the client proxy. It can be in the user's computer (lightweight), in an organization outside/with the firewall, or even in the ISP office. It is used as a proxy. It should have a large disk space for caching. It should work with all protocols. It should have FEC. It should load when the computer boots. It should have a GUI for configurations.

I think that this proxy should be HTTP 1.0 to the client, but Edan thinks differently. When working with HTTP 1.0 the browser will open several concurrent connections, rather than using the same connection for several resources. I think that the first method is more efficient, because it allows us to return any of the resources and return others. The advantage about not creating TCP connections disappears because the proxy and the browser are near (on the same computer or LAN).

It will be free. I will buy everyone a drink when we will have 1000 users who downloaded the client ☺.

:TBD: Describe it a little more.

6.8. User Agent (Browser)

This is the user browser, or whatever. We should configure it to use our client as its proxy.

7. Protocols

There are 6 new protocols. Each of them needs different resources, and has its advantages and disadvantages.

	With Content-Group-Header	Without Content-Group-Header
Multicast	FEC: Yes Code: 50:50, random, iid Differential: Yes (common files) Pre-fetching: Yes (content-group) Flow: RLC Init: By Statistics Center Client: Does not send data	FEC: Yes Code: Mostly systematic Differential: No Pre-fetching: No Flow: RLC Init: By Statistics Center Client: Does not send data (Need to send some side-information packets all the time)
Reliable UDP (Unicast)	FEC: Yes Code: 50:50, random, iid Differential: Yes (common files) Pre-fetching: Yes (content-group) Flow: Client sets speed Init: By Client, UDP Client: start, stop, and set-speed.	FEC: Yes Code: Mostly systematic Differential: No Pre-fetching: No Flow: Client sets speed Init: By Client, UDP Client: start, stop, and set-speed. (Need to send some side-information packets at start)
HTTP as Transport	FEC: Yes Code: 50:50, random, iid Differential: Yes (common files) Pre-fetching: Yes (content-group) Flow: TCP, retransmissions Init: By Client, HTTP Client: HTTP & TCP protocols.	Uses the old HTTP (no-FEC, no-code, not differential), or, sends Content-Group-Header, and then the data, like the block on the left side.

Table 1 – Basic Protocols

7.1. Basic protocols

7.1.1. Multicast, with Content-Group-Header

This is "the" protocol. The whole system is build in order to get to this block. At this block, the clients do not send even one IP packet to the server. This means that the server can handle any number of clients, using a very simple data carrousel.

:TBD: We need to define the application-level interface to this protocol.

7.1.2. Reliable UDP, with Content-Group-Header

When no multicast addressed available, or in a non-multicast region of the Internet, the most efficient protocol is this one. It uses the Reliable-UDP protocol that we shall define, with the ability to add equations for files that the client already has.

:TBD: We need to define the application-level interface to this protocol.

7.1.3. HTTP as Transport layer, with Content-Group-Header

Sometimes clients cannot use the UDP protocol. For example, a client behind a firewall, a PDA, or a **HAVi** device might not have UDP access. In these cases, we can still have the pre-fetching and the differential downloads using HTTP. We simply put the data packets to HTTP connection, and the client reads until it has enough. I hope that we will not need it, because Reliable-UDP is more efficient, and I guess that it will take fewer resources in the server side.

7.1.4. Multicast, without Content-Group-Header

This is what Digital Fountain do. It is good just for one thing: send big files to a lot of users. It is feasible to use Java Client in order to collect the file and store it on the user's hard drive. This format is not suitable for sending small files, nor for sending HTML pages.

:TBD: **Nadav, Meir**: We cannot use systematic equations and RLC. How does Digital Fountain code works? How can it handle the flow-control?

7.1.5. Reliable UDP, without Content-Group-Header

The reliable UDP can be used as a transport layer, like TCP. There is a major difference, however: TCP (like all other layers in OSI and other communication

models) is stream-oriented. When TCP connection is established, the sender has a pipe, which it can put data into and the receiver will get it. The reliable UDP, on the other hand, works in the resource level, when a resource is an array of bytes who is known to the server when opening the connection. The protocol I suggest here does not support compression, sub-files, or adding equations for parts of the file, **meaning that we cannot use it for downloading the Content-Groups**. This protocol fits only in the block that I called "Optional reliable-UDP for resources", and for downloading resources using a download-client, like **Digital Fountain**. We can still use it effectively.

The server uses **listen()** for a given port number and IP, as always. The client initiate the connection by calling a non-blocking method:

accept-file(server-IP, server-port, client-port, initial-rate, URL)

Than, the client sends a single UDP request packet to the server (and some re-transmission mechanism if this packet is lost). When the server gets this packet, it should check if the URL exists. If not, it will send a single file-not-found notification packet to the client port. If it does exists, it will load it (or map it) to memory, and call the non-blocking method:

send-file(client-IP, client-port, URL-data, initial-rate)

Then, the packets will travel from the server to the client, at the initial rate. The client should send ACK command from time to time, DONE command when it has enough packets, CHANGE-RATE command when there are too many or too few loses, and so. We should use some of the ideas in the RLC code ☺. The server should terminate the connection if the client sends DONE, or does not send any packet for some time.

The server should start by sending a packet that contains this information:

File Header – A 64-bit header, as defined above.

U32 – The size of the resource, or –1 for file-not-found.

U8[20] – The SHA of the resource.

U8[*] – The URL, as sent to the **accept-file** command.

In order that this information will not get lost, we can, for example, send it 3 times to the client, or, add a re-send command for it (while holding the packets until we get it without being able to decode them). The client needs this information for several reasons:

1. Version control. This ensures that the client and the server are using the same version of the FEC and so.
2. Size of the resource – in order to know how many packets it contains. This is critical for the decoding.
3. Verifying that both sides are talking about the same resource.
4. Verifying that when the transfer ended, the client has the correct file.
5. Allowing a smart re-connect, or **resume** option in the client ☺.
6. Allowing the client to detect that it already has the file, and abort the download.

The above list implies that the client should get this header immediately, and have the opportunity to allocate buffers, or deny the download (if too big, or if already exists).

Only when the connection is ended the server is able to de-allocate its memory buffer. The client should get a notification at this stage, with the header information and the suggested bit-rate for next time. Only then, the transport layer can free the resources.

:TBD: Maybe we should add the differential download to this protocol? This should be relatively easy. Then, the client should be able to add equations from the previous version of the file it has. For example, the client will send the SHA of the version it currently have, and the server will send a cut & paste command for adding equations. This can be real nice feature! This is not **rsync** – the server should know the exact version that the client has from its SHA.

7.1.6. TCP as Transport, without Content-Group-Header

Same as above, but when we do not have the Content-Group-Header. The solution is simply send it first (which works, because TCP is reliable), and then put the encoded packets. I hope that we will not need it, because Reliable-UDP is more efficient, and I guess that it will take fewer resources in the server side.

7.2. Tricky protocols

7.2.1. Stateless Reliable-UDP

In general, the Reliable-UDP is a statefull, complex protocol (although my guess is that TCP is worse). The server should remember each client's IP and port, which resource, the rate, RTT approximation, and much more.

But, we can use a little trick in order to send small files. We need it, in order to send Content-Group-Headers to the client. For example, the server wants to send a very large Content-Group-Header, which takes 1800 bytes. Let's say that we use 512 bytes per packet. The server will break the file into 4 packets, and encode them into 7 packets. Then, it will put all the packets on the Internet.

There is a good chance that at least 4~5 of these packets will reach the client. If they did – we have saved a huge amount of resources in the server. If they didn't, the client will ask again, and the server will send another 7 packets. Any 4~5 of these 14 packets will allow us to decode this resource.

I think that even in this second case we have positive savings. Starting a protocol, sending packets, waiting for all the packets to get, and then send a termination packet (which might get lost too) is much more complex (and takes more RTT) than this stateless protocol. :TBD: Meir, Nadav: Do you think that it's worth a patent?

7.2.2. Combined protocol

:TBD: Define exactly.

:TBD: Move it to another place in this document.

I thought about some kind of optimal mixture between the protocols, which the statistics center decides on. The idea is simple.

First, the client checks if it has the Content-Group-Header or not. For the most common Content-Groups it will have it, because the server will push it to the client. If it has it, it should try to access it using multicast, then, fallback to unicast, then, fallback to HTTP transport, and finally to the old HTTP protocol. Please notice that the packets in the first 3 protocols are the same. The decoder can get packets from all of them, and use them in the decoding process.

On the other hand, if the client does not have the Content-Group-Header (or any other information), it will send a single packet to the server using Reliable-UDP. This packet will contain the needed URL, as defined above. The server may:

1. Send a single packet, notifying that this file is not accelerated at all. In this case, the client will use the old HTTP for it. This notification may or may not contain the SHA of the resource in that URL. If it does contain SHA, and the client has a file with this SHA, it can use it.
2. Send the requested file using Reliable-UDP, as defined above. The first packet will contain meta-information about the file SHA and size, so the client can abort the download if it already has this file.

3. Send a single packet, notifying that this resource is transmitted using multicast, and give the IP address of the multicast group. In this case we can also send SHA and size. This allows us not to send meta-information when sending the packets in multicast. :TBD: Should we use it?
4. Send a single packet, notifying that this resource should be accessed using encoded HTTP which contains the Content-Group-Header, as defined above.
5. Send a Content-Group-Header that contains the file using Reliable-UDP. Than, using this Content-Group-Header the client will execute this combined protocol from the start. Since this file is small, we can use the "Stateless Reliable-UDP", as defined above.

I think that this is simple and efficient way for transporting data based on the Statistics center.

8. Building blocks

8.1.1. SHA-1 (Secure Hash Algorithm)

SHA-1 is an open-source digest algorithm from NIST. It accepts an array of bytes (actually, bits), up-to 2^64 bits long, and convert it into 160 bit signature. This algorithm is highly efficient for software implementation, since the 160-bits are actually 5 32-bit registers, with only register level operations between them.

This algorithm, unlike CRCs and so, is cryptographic-secure. This means, that one cannot feasibly find a file that their SHA-1 digest is X. One cannot create two files with the same digest. There are (almost) no collisions. We need all these features of the SHA-1 algorithm.

SHA-1 is superior to the obsolete original SHA algorithm, which had a flaw. In all places that I say or write "SHA", I mean "SHA-1".

8.1.2. FEC (Forward Error Correction)

A FEC is an Information-Theory base technique, which allows to send encoded information, and recover it in the client, without specific requests for retransmission. I will give a short description of it, without getting too deep.

We want to transmit a file. First, we break the files into **N** blocks. Each block is exactly in the size of the IP packets that the transmitter will use (maybe 512 or 1024 bytes). The encoding process uses a seed for pseudo-random bits, and generates N bits, corresponding to the **N** blocks. Then, it uses bit-wise **xor** on the blocks, which the bit is set. The resulting block is sent over the Internet, with the seed, and a checksum. Then, it does it again, infinitely.

The receiver collects packets. It validates the checksum. Then, it calculates the N bits from the seed. The receiver collects **N+e** packets, until the matrix with N column and **N+e** rows is in the rank of **N**. Then, it can inverse the matrix, and resolve the code.

We can sh[redacted]at if the bits are **iid**, with 50% to be one, [redacted] expectation of e is about one. This means that in the normal case all we need is one extra packet in order to decode the data.

The client needs some side information. It needs a version – to validate that the transmitter and the receiver uses the same FEC. It needs the SHA of this resource in order to validate it at the end. It needs the size of the resource in order to know N, and in order to build the file without padding zeros.

The real world is a little more complicated. We need the decoder to do the job sequentially, which means, that it should do some small amount of work each packet, rather than waiting to all the packets and then blocking the computer.

Sometimes the client has some of the information in the encoded file. For example, when this file is a Content-Group and the client have some of the files. Other example can be any form of differential downloading (:TBD: Specify methods for it). In these cases, the client should be able to add systematic equations to the decoder. These equations should be handle efficiently.

It also might be able to get some of the blocks before others. We need each block immediately. This will enable the application layer, using Content-Group-Headers, to decode some files before others. I might need this feature for a new idea that I have. :TBD: Write it.

:TBD: We need to define the exact interfaces of the encoder and decoder to support all these requirements.

8.1.3. Field size, byte order

In order to support interoperability, we shall not use C++ **int, short, long**, and so. These are not well defined in two aspects: one, is the number of bits in them, and another is the byte order (little/big endian). In this document, we will use:
U8 – Defined to be unsigned 8-bit integer (0 to 255).
U16 – Defined to be 16-bit unsigned integer (0 to 65535), in the Network byte order.
U32 – Defined to be 32-bit unsigned integer, in the Network byte order.

All the file formats are in the Network byte order, formerly know as Motorola byte order (MSB first). We shall not use floating-point numbers inside files. All file formats should be defined such that each data size is aligned to its boundary. This means that **U16** should be at even-bytes offsets and **U32** in quad-bytes offsets.

The **ACE** package has the same definition in different names. For the implementation, we shall use ACE. However, we need support for both streaming (a read command which swaps the bytes if needed) and memory-mapped files.

8.1.4. File (and block) headers

All the files (and blocks inside the files) that we use shall have a common 64-bit header. This will allow us to nest blocks in a robust way. This header should be aligned on 64-bit boundary on all circumstances. The header is:

U16 – Type of file. We shall use a common definition file for the whole system.

U8 – Major version number.

U8 – Minor version number.

U32 – The size of this block or file, including this header.

Every program must validate the type of file it reads. If it does not match to the expected type, the program must halt, and report it.

In general, the major file version is for incompatible changes, and the minor is for compatible ones. When a program that designed for a specific version reads a file, it should check the version. If the major version is bigger than expected – this file cannot be read. If the major version is smaller than expected – the program should decide if it wants to convert it or not (backward compatibility). If the major version is as expected, the program must read the file correctly, and the file should contain information depending on the minor version number.

For example, version 1.0 defines a bit-field, which includes some reserved bits. A writer for version 1.0 must set them all to zero. A reader for version 1.0 must ignore these bits (forward compatibility). Let's suppose that version 1.1 give meaning to some of these bits. A writer for version 1.1 must set these bits to their correct values.

When a reader version 1.1 read version 1.0, it should understand that these bits do not have a value (backward compatibility). In order to achieve this the easiest thing to do is usually to convert all previous versions to the latest version. This is easy to implement with streaming, but a little harder for memory-mapped files.

The block size is needed in order to skip known or unknown blocks, and integrity checks.

8.1.5. Resource

A resource is single file on a disk, byte array, or whatever you call it today. The name of each resource is its 160-bit SHA digest, which depends only on its content. It's name uniquely defines it's content, for all practical purpose.

A resource does not have a file-extension, a mime-type, a date, or any other meta-information about it. These will be added, if and when needed, by higher level definitions.

8.1.6. Compression

Compression here is a process that maps any byte array to any other byte array, which is usually smaller. The reverse mapping called de-compression. The compression works on file basis, like ZIPs, unlike CABs.

The compression **MUST** be deterministic, and well defined. For any given uncompressed file, there is a single compressed file. It sounds trivial, but ZIP compressors, in general, do not do it! For example, you can tell the ZIP compressor to use normal compression or extra compression. Without being well defined, we will fail to implement the erasure code! This is very important, and should be compatible between versions of our product.

We can define that the first byte is the compression type, or zero for uncompressed resources. However, this still means that if one encoder decides that the compression method will be X, all other encoders must reach the same decision.

Since this is critical, as a safeguard, we shall add a checksum of the compressed resource in order to validate that this actually works. A simple 32-bit checksum should be enough; there is no need for a secure digest.

8.1.7. SHA Cache (for the client)

The cache, as seen from outside, is a **hash-table** of **compressed resources**. For each resource the cache holds:

1. The name of the resource (which is the key to the hash-table).
2. The U32 checksum of the compressed resource.
3. The compressed resource itself.

It is important to note that the cache does not hold the original URL, mime-type, extension, date, and so. This spec will never be coherent if it does. The cache holds some internal information for the LRU algorithm, or whatever we may implement, but there are no set and get methods for them!

The resource-level **operations** on the cache are:

1. Get the uncompressed resource data by its name – as stream to socket.
2. Get blocks from the compressed resource data by its name – for the decoder.
3. Get the checksum of the compressed resource by its name – for the decoder.
4. Store a (new) resource; Fill all the fields, and validate that the digest of the (uncompressed) resource and the checksum of the compressed resource match their expected values.
5. Put a listener that waits for a given resource to appear and then sends a message to some thread.

We need to store the compressed form in order to retrieve it when needed for the erasure code decoding. This saves the need to put the compression code in the client. We might need to implement a second uncompressed cache if the client asks for a resource too many times, and we do not want to decompress it each time. My guess is that we can implement the decompression on-the-fly when writing to the output socket.

Other operations are for management only. Examples are create, destroy, compact, empty, set max allocated size, and so. There are no operations like enumerate, invalidate, update, remove, and so. We do not need them ☺.

8.1.8. Content-Group-Header

A Content-Group is a set of compressed resources. The Content-Group-Header is a single file, which defines the resources inside a Content-Group.

The file format is:

File Header – A 64-bit header, as defined above.

U16 – The erasure code block size that this file was designed for (512 bytes?).

U16 – Number of resources described in this file.

For each resource:

- **U32** – The size of the compressed resource, including the padding of zeros.
- **U32** – The size of the compressed resource, without the padding of zeros.
- **U32** – Checksum of the compressed resource. For compression validation.
- **U8[20]** – The name of the resource.
- **U8[?]** – :TBD: Meta-information, which includes mime-type. Variable size?

Each Content-Group-Header has a name, which, as always, is defined to be the SHA digest of it. Please note that since the header contains the SHA of the resources themselves, this name also defines the whole Content-Group. One cannot create two valid Content-Group-Header names with different content.

:TBD: We may need to inc[redacted]some kind of priority bit for the first res[redacted] in this Content-Group. If this bit is set, it means that asking for this resource will cause, with high enough probability, requests for all other resources in this Content-Group. This means that if this first resource is required, the client should prefer this Content-Group above any other.

8.1.9. Content-Group-Data

A Content-Group-Data is a single file, which is the Content-Group-Header, joined with the compressed resources and zero bytes padding, as defined by the header. The name of the Content-Group-Header defines the content group uniquely.

The Content-Group-Data exists only in the server. The client never gets this file as-is. For this reason, there is no need to give it a name depending on this file SHA digest. Using the Content-Group-Header name is sufficient.

8.1.10. Validation of Content-Group

Validation of Content-Group is a command to the cache, which gets a Content-Group-Name, and returns a Boolean, which is true if the Content-Group was validated correctly.

All the definitions in this paper leads to one thing: To create a **bullet-prove** mechanisms that will let the client know, for a 100%, if the return value should be true or false. And if it's false, this process can detect, with 100% accuracy in both directions, which files the cache already has. Later, we can consume bandwidth only in the sum of the size of these files. This is a case when technology can solve problems before they are even created ☺.

When a client gets a Content-Group name, it can search the cache for an entry with the given name. If it finds it, it can validate the digest of it. Then, it can access all the relevant files in the cache, and validate their digest too. When validating these files, it can detect errors that the simple checksum missed, and throw some kind of internal error exception for compression errors. If all files exist, the receiver can, in principle, create the Content-Group-Data itself.

8.1.11. Encoded-Content-Group

This file, if needed, contains a small header, and then a large number of data packets. It exists only in the server side. :TBD: We need this file format if we want that the UDP server will have a simple job at runtime, something like a data carousel. I am not sure about it. :TBD: **Nadav** should help me define the exact parameters inside it.

As far as I can tell, each packet of data will contain at least these fields:

1. The encoded data itself, as defined in the block size above.
2. A **U32**, which is the seed to the random-bits generator that created the code of this block. This will allow us to send a small number of bits and describe a whole row in the code matrix.
3. A **U32** checksum of the Content-Group-Header name, the seed, and the data. The receiver can check that this block received correctly.

The last checksum allows us to detect transmission errors in this packet, since the UDP, as far as I know, does not have a CRC field. Also, and more important, it can detect misconceptions about which stream this packet belongs to. This is important, because the current suggestion of the implementation of the multicast in the Internet is lousy, and might cause one client to accept packets which were used by another group.

9. Old document (will be removed)

:TBD: Rewrite everything from here and below.

:TBD: Place these

U32 – IP address for HTTP or -1 for the current IP. Zero if not available.

U32 – IP address for unicast or -1 for the current IP. Zero if not available.

U32 – IP address for multicast group. Zero if not available.

9.1.1. Downloading Content-Group from the Internet

In order to download a Content-Group, all the client needs is the Content-Group-Header, and, maybe, some base IP (the IP of the active host). In the header there are 3 IP fields, which allows the client to download it using up-to 3 different protocols. The content provider should decide which protocols are valid for each Content-Group. If more than one way is given, the client should try the multicast first, then the unicast, and finally use the HTTP. In all cases, there is no need to re-download the Content-Group-Header itself. Only the data is needed.

Failure occurs when:

1. The IP address is set to zero.
2. The protocol was disabled in the client.
3. The server does not respond.
4. The Content-Group validation process fails after the download.

When using HTTP protocol, the client needs IP address, port, and path. If the IP address in this locator is 0, the Content-Group cannot be accessed using HTTP. If the IP is -1, the IP is taken from the original URL that connected us to this host (we need

to define the exact semantics for it). Otherwise, this field contains the IP address to use. The port is always 80. The path is some prefix combined with the Content-Group name. For example: **/__UltimaCast1/resource/012...DEF**.

Please note that the HTTP option is inefficient for a Content-Group that includes many resources, because if some resources exist in the client cache, it still has to download them. We may overcome this limitation using HTTP 1.1 protocol.

In order to get the Content-Group using unicast, the client needs an IP address, as defined above. Then, it can access the server somehow, and ask for the Content-Group. We need to define the details of this process, but I believe that an IP address and Content-Group name should be sufficient.

In order to get the Content-Group using multicast, the client needs an IP address of multicast group. The rest of the erasure code decoding information exists in the Content-Group-Header. We might need some extra information for joining the multicast group, depending on the actual implementation of it. For example, the 64-bit group key for private groups.

Each downloaded resource is verified to match its name and its compressed checksum. Than, each (compressed) resource is stored separately in the cache.

Before the download process begins, the validation of Content-Group process is executed. If it succeed, no further work needs to be done. If it fails, the cache is queried for individual resources. Each located resource adds equations to the decoder, using the get-block-from-compressed-resource cache command. This is one of the most important features of this system. We should patent it, ASAP.

9.1.2. Host-Master-Content-Group

Each host has, at any given time, a single active Host-Master-Content-Group. This is actually a Content-Group, with one major addition: the resources inside it are all the Content-Group-Headers that this server serves.

It is important to understand that this Content-Group-Header name defines uniquely, recursively, all the accelerated information that the host has at this moment! In order to get this information, however, there are several levels of indirection, which actually allows do download the information differentially and on-demand.

9.1.3. Obtaining the Host-Master-Content-Group

:TBD: Support HTTP redirect here.

First, the client needs to obtain the Host-Mater-Content-Group-Header. In order get it, the client connects to **http://host/__UltimaCast1/active**. If this URL does not exist, this host is not UltimaCast 4 WWW accelerated host. If it exists, it will contain a file in this format:

File Header – A 64-bit header, as defined above.

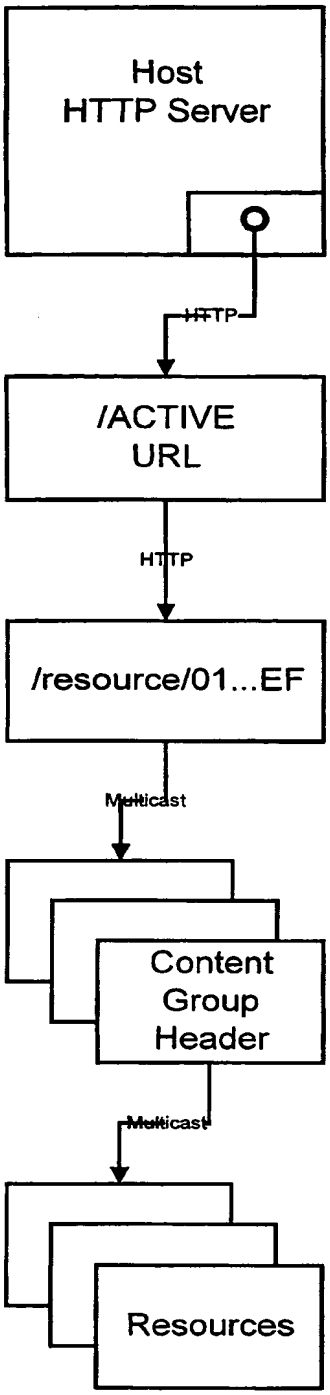
U32 – General flags, describes how the server was created, and so.

U8[20] – The name of the active Host-Master-Content-Group-Header.

The client has to perform several steps. First, the client should download this data, and remember it. Second, the client needs to check if it already has this entry in its cache. If not, the client should download it using the normal HTTP protocol from the normal HTTP URL, and store it in the cache, for the next time. The amount of data is expected to be very small – about 50 bytes per Content-Group. This gives us about 5KB for a big host. The client should remember it in-memory.

At this stage the client still does not have the required information to work. The client needs the Content-Group-Headers themselves. So, third, the client enters the download of Content-Group process, as defined above, for this Host-Master-Content-Group. Normally, this should be done over multicast or unicast. This means, that if there are, for example, a 100 Content-Groups, and only 5 of them were updated from the last time, a bandwidth will be allocated only for those 5. Each downloaded Content-Group-Header will be put in the cache, as always. This information too should be stored at the client memory.

When this 3-stage process ends, the client is ready to do its job as a proxy. It has all the Host-Master-Content-Group in-memory. The size of this structure is about 100KB to 300KB for a big host. At this stage, the client can (almost) answer a single type of request that the browser asks it from time to time: give me the HTTP data for a file with SHA digest X.



The client will access it using HTTP protocol.
Not cached - need to be accessed each session.
Takes fix size of about 50 bytes.

The client will access it using HTTP protocol, at the beginning of the session, if needed.
Takes about 5K.

The client will access all these using Multicast, at the beginning of the session, if needed.
Takes about 100 * 1K in the first time, and works differentially later.

The client will access all these using Multicast, when needed by the browser. Works differentially.

Figure 2 - Information Hierarchy

9.1.4. How to know the SHA of the needed resource?

:TBD: Rewrite it.

:TBD: **Omer, Edan** and **Yuval** thinks that we should never change the original HTML. **Yuval** even though that maybe we will fallback to normal HTTP, and renaming resources will cause the browser to fail. I have to agree.

:TBD: Define exactly how we shall get the SHA.

In order to search for a resource, we need an a-priory knowledge about its SHA digest. There are several methods to know it. We can use all of them together, constructing an efficient and robust system.

Method one: Change the name of the resources. Instead of linking to **myfile.jpg**, the HTML page will link to **sha_012...DEF.jpg**. The client will detect URLs which their file part has this format. When such a URL is detected, the client will know the SHA of this file. It can also guess the mime-type from the extension. We actually do not need to really guess – we can define a strict and expandable semantics for it. For example, we can agree on a table of known extensions and their corresponding mime-types, and allow a format of **sha_012...DEF.application_x-zip**, for the "application/x-zip" mime-type.

:TBD: **Yossi** suggested adding a tag in the HTML which gives the SHA of resources. We can parse it, and detect it. I think that it will be too intrusive, and will take too much CPU on the client, but I need to check.

The above method is the most efficient and secure way we have. The problems that this method creates are non-technical. In order to use it we have to change the original pages in the site, and change the URL that the user sees in the browser, if we want to accelerate HTML files. Still, we should use this method wherever possible.

Method two: For each host[redacted]a URL that will tell the SHA values an[redacted]me-types of all others URLs in the server. We need to define the exact semantics. For example, accessing **http://host/__UltimaCast1/query/X** will return the SHA and mime-type of the URL in **http://host/X**, when X is any valid path. In this way we need extra round-trip per resource, but the server can return accurate results.

Method three: Add some pre-loaded structure that maps from URL to SHA digest and mime-type. In this way we do not need to change the server pages, and we do not need round-trip per resource. We do, however, need to push some data to the client, and enter into all the dirty problems of coherency we are trying to avoid ☹.

9.1.5. Give me HTTP data for file with SHA digest X query

Even at this stage, this is not as simple as it sounds. The client still has to give some meta-information on the file, like it's mime-type. Without the mime-type the browser will not work. Please notice that the mime-type is **not** a part of the file. This means, for example, that two sites may use the same file with different mime-type to cause different plug-ins in the browser to process the file. It is preferred that we support this feature, because it can save us from getting into trouble in the future.

So, there is no way to support the query that defined above. But there are several ways that we can approximate it close enough.

If the file name is using the **sha_012...** convention, and it exists in the cache, we can immediately return it, without any additional information. This is a highly efficient bypass. It even has extra benefit: It can find resources that exist in the cache, but the current Host-Master-Content-Group does not include them anymore.

The normal method is a little more complex. The client need to know the whole URL, and use it in order to get this information:

1. Which is the current Host-Master-Content-Group, and general flags?
2. What's the SHA of the file? This can be done using any of the method defined in the relevant section above.

3. There is no need to get the mime-type from the URL. It is critical, if we do not want to modify the original HTTP server.

Then, the client need to search the Host-Master-Content-Group for files with the given SHA digest. If it doesn't find any, it should fallback to normal HTTP, since this resource is not accelerated.

If it finds it exactly in one Content-Group, it should download this group. This should not be overkill, since the Content-Groups are designed for it. For example, let's assume that user want to see the page **x.html**, which is not accelerated, but embeds the accelerated images **1.jpg** and **2.jpg**. We can decide to put both images into a single Content-Group. Asking for **1.jpg** will cause **2.jpg** to be downloaded too, but it's OK since we believe that the browser will ask for it eventually. We get pre-fetching for free ☺.

We might, however, find more than one Content-Group that embeds the file with the given SHA digest. One case is when this is the first file in the Content-Group, and it marked using the priority bit. This means that after returning the data of this resource, the browser will surely ask for all others in this Content-Group, and it is better to download this Content-Group right now.

Another case is when this file is a normal resource in more than one Content-Group ⊗. This should not happen a lot. In this case we can:

1. Pick one group by random.
2. Pick the smallest group.
3. Pick the group with the best protocol.
4. Pick the group which most of it is in the cache.
5. Wait for a clue from the browser.

The wait-for-a-clue method is the harder to implement, but will give the best results. We do not have to implement it in version 1 of the client. In this method, we should wait for a while, for the browser to ask for another resource. For example, the image **1.jpg** exists in several Content-Groups, and we cannot decide which one to chose. But, after 10ms, the browser asks for **2.jpg** using a concurrent TCP connection in a different thread. At this stage, we find that there is only one Content-Group with these two files, and it does not contain extra files. We should pick it.

This is hard to implement. There is a huge difference between something that I will call a "transactional" multi-threaded program than this one. Normal servers look to the outside world as-if they work in transactions. When they are two concurrent connections, the server can be viewed as if it does one of them first and the second one later. Here, **this is not the case**. Getting query for only **1.jpg** would cause the client to download **CG1**. Getting query for only **2.jpg** would cause the client to download **CG2**. But getting both caused the client to download **CG3**.

This is the only place in the client that we need heuristic, multi-threaded algorithms. It is needed only if the main HTML is not accelerated. If the main page is accelerated, we will not get to this kind of decisions too much.

9.1.6. Differential downloads

:TBD: I need to define it. This is **not** in the scope of the first version.

:TBD: Can this be patented?

:TBD: Compare with **rsync**.

We can use the above mechanisms for highly efficient differential download. The basic ideas are these:

1. We break a single file into several parts. We describe each part using its SHA digest. Then, we multicast it (or unicast it). The client, as always, can use extra equations if it has some of the parts. The parts are, as always, byte-oriented.
2. We can send a suggestion to the client, which state: If you will take file with SHA digest of **X**, and crop the part which starts with offset **Y** and length **Z**, you will get a file with SHA digest of **W**.

For example, let's say that we want to multicast the file **x2.bin** to some clients. We know that some clients have **x1.bin**, which is the previous version of this file. Let's assume that the offset **off1** length **len1** in **x1.bin** is in offset **off2** in **x2.bin**. We can tell the clients that if they will crop the file **x1.bin** from offset **off1** length **len1** they will get a file with SHA of **sha1**. Then, we will send a description that tells that the file **x2.bin** is a concatenation of files with SHA of **sha2, sha1, and sha3**, at this order. Clients with **x1.bin** can use extra equations, and download this file more efficiently.

This method is very simple, and efficient, but it can only add equations if the common part is longer than a packet or two. With this limitation, we can almost find the optimal way to divide a file into parts ☺.

Please note that this method can be used when several files share the same data, or when a file has a copy of the same data twice. We can do a lot with these two simple commands, of crop and combine.

9.1.7. Changes required for partial multicast deployment

The Internet will not become multicast at once; maybe it will never be one big multicast area. Instead, we will see bigger and bigger "islands" which are multicast enabled. Each of these islands will allocate multicast IP addresses of its own, using some protocol inside it (Maybe all islands will agree on some version of IGMP, but its not required). Normal IP packets will be the bridges between islands.

We hope that the UltimaCast 4 WWW system, or a future UltimaCast-2 system, will be that bridge. I will start with an example: Client **C**, which is inside island **J** wants to connect to host **H**, which is out of **J**. The host **H** should tell **C** about another host, **P**, which is inside **J**. Then, **C** should get information from **P** using multicast. When **H** is updated, it should also notify **P** about this update, since it needs to reflect it. This method can be used as an implementation of a distributed network load balancing ☺.

This document describes the static part. It does not describe the updates in the server, how to allocate multicast addresses, how to send statistics from the client, and so. In the scope of this document, only minor changes are needed to support this new concept.

In the section about obtaining the name of the Host-Master-Content-Group, the client should connect to a fixed HTTP in the host, and get fixed amount of data. Now, we need this data to be dependent on the client IP address. The HTTP server should check the IP address of the connection, detect the nearest server, and return the name of the Host-Master-Content-Group for it. We should add an IP address, in order that the client will get the Host-Master-Content-Group from this server directly (using HTTP, as you remember). So, if two clients from different islands connect to the same server, they will get different Content-Group names, different data inside it, which will direct them to different IP addresses for the multicast groups.

If there is no load balancing, and each client is connected to a single host, the change above is the only change we need. But, if, on the other hand, the same client can be redirected to different hosts for the same content, we need to adjust things a little. In the suggestion above, the IP addresses for obtaining the Content-Group are defined inside the Content-Group-Header. This means, for example, that if a client connected to a host, and got all the 100 Content-Group-Headers (which takes about 1K each), when connecting again to a different host (because of load balancing), it will need to download them again (since the SHA digest of them changed).

In order to solve this inefficiency, we should move these IP addresses into an upper-level of indirection. Instead that each Content-Group-Header will have a single set of IP addresses in order to retrieve itself, it will have a set per resource that it contains. When connecting for the first time, this change will not effect the amount of data that the client needs to download. However, when updating the Content-Groups that the client supports, it will be easier to change the IP addresses for the resources, resulting a more dynamic system, which is capable of load balancing. The price is, for example, when we add a Content-Group the server will need to send, for each of the 100 content groups, the IP addresses with the SHA. This might double the size of this message. Since this message is small anyway, this is a small price to pay for a more dynamic system.

Please note that this means that the Host-Master-Content-Group is not a Content-Group anymore, because it has IP fields inside it. This is a minor difference. I will update it in a future version of this document, because I want you to review it as the hardcopy you have.

EXPRESS MAIL NO
EL 510314798 4S